OPTIMIZING PARALLEL SIMULATION OF MULTI-CORE SYSTEMS

A Dissertation
Presented to
The Academic Faculty

by

Zhenjiang Dong

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2016

OPTIMIZING PARALLEL SIMULATION

OF MULTI-CORE SYSTEMS

Approved by:

Dr. George F. Riley, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Douglas M. Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Richard M. Fujimoto
College of Computing
Georgia Institute of Technology

Dr. John A Copeland
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: November 20, 2015

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor Dr. George Riley for his help and guidance throughout my graduate study. I would not be able to finish my Ph.D. study without his guidance and help. I would also like to thank my dissertation committee members, Dr. Sudhakar Yalamanchili, Dr. John A Copeland, Dr. Douglas M. Blough and Dr. Richard M. Fujimoto for the valuable advices and suggestions.

At the same time, I would like to thank my beloved wife Hanying Zhang and my parents Xiaoju Zhang and Minghua Dong for all the unconditional support encouragement and love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

LP	Logic Process
NS3	Network Simulator 3
FNM	Forecast Null Message
DES	Discrete Event Simulation
PDES	Parallel Discrete Event Simulation
CMB	Chandy-Misra-Bryant
LBTS	Lowest Bound TimeStamp
VLSI	Very Large Scale Integration

SUMMARY

The simulations of multi-core systems are widely used by both researchers and industry computer architects to verify their design before implement actual products. Due to the complexity of multi-core systems, traditional serial simulation for such systems can be time consuming. Therefore, parallel discrete event simulation has been employed to reduce the time requirement and achieve scalability for the simulation. For parallel discrete event simulation programs, optimizations for synchronization algorithms, partitioning schemes and other aspect can be done to improve the performance and parallel efficiency. The objective of this dissertation is to design, develop, test and evaluate a variety of technologies to improve the performance and efficiency of parallel discrete event simulation of multi-core systems. The technologies include a general guide for partitioning schemes, an efficient front-end for timing-directed simulation, and a new conservative synchronization algorithm. With the technologies, our simulator achieves competitive performance against other parallel simulators for multi-core systems.

CHAPTER 1

INTRODUCTION

Multi-core design for CPU is the recent trend [1] and we believe this trend will continue in near future. Researchers and industry CPU architects utilize simulation to evaluate their designs and gain a certain level of confidence before manufacturing the actual products. So, simulation plays an important role in field of multi-core system design. Due to the fact that modern multi-core systems are complex [36], traditional sequential simulation can hit the bottlenecks in terms of execution time [2]. To handle the complexity and achieve reasonable scalability for simulation, [3], [4], [5] and [6] proposed parallel simulation for multi-core systems with parallel discrete event simulation (PDES) [7]. PDES programs run on parallel computers can lead to efficient execution of large simulation programs by providing methods to utilize hardware resources in parallel [7].

In PDES programs, simulation is separated into processes called logic process (LP), and LPs can utilize independent hardware resources. During parallel simulation each LP generates and processes events in a distributed manner by sending and receiving messages with time stamps. Synchronization algorithms are employed in PDES programs to ensure the global order of events and correctness of execution. The task of synchronization algorithms is generally complicated because the LPs can operate on different Local simulation time while the global order must be maintained. Poorly designed synchronization algorithms can lead to equal or even worse performance than the sequential simulation.

The partitioning schemes determine how to partition the parallel programs into separate LPs and how to assign each LP across the parallel computer. Together with the synchronization algorithms, partitioning schemes can also have great effects on the performance and efficiency of the parallel simulation. Therefore, well designed synchronization algorithms and partitioning schemes should be applied together to take advantage of parallelization.

1.1 Contributions

The objective of this dissertation is to design, develop, test and evaluate a variety of technologies to improve the performance and efficiency of parallel simulation of multi-core systems. The technologies include a general guide for partitioning schemes, an efficient front-end for timing-directed simulation, a new conservative synchronization algorithm.

- We study the effect of partitioning schemes can have on parallel simulation of multi-core system. From the test results we summarized that the partitioning schemes can have significant effects on the performance and scalability. Additionally, the time that consumed by the components have increasing number of inter-LPs links with growing system scale to gather and process the null-message is the major reason that leads to the differences. In null-message based parallel simulation, such components should be partitioned when the system scale reached a certain level to achieve reasonable parallel performance.
- The timing-directed simulation of Manifold project [3] use a front-end called Qsim. It responsible for supplying the instructions to the architecture components of back-end during simulation. To achieve reasonable parallel performance and scalability the front end must be efficient. However, with original design the Qsim [28] front-end alone cannot operate with Manifold components in full bandwidth and becomes the bottleneck of the performance. To handle this problem, we designed and implemented an intermediary front-end called Proxy. The Proxy helps to better utilize the bandwidth of Qsim front-end and greatly improved the performance.
- To improve the performance of null-message based parallel simulation of multi-core systems. We designed and implemented an enhanced null-message algorithm which we called Forecast Null-message algorithm (FNM). It is different from traditional null-message algorithm that has only the application-independent optimizations, it utilizes the domain specific knowledge that acquired from architecture components to improve the lookahead of null-message and significantly improved the overall performance.

1.2 Dissertation Organization

The rest of dissertation is organized as the follows. In chapter 2, we briefly discuss the background and related works done by other in similar area. In chapter 3, we present our study for the effect of partitioning schemes have on null-message based parallel simulation for multi-core systems. In chapter 4, we discuss the Proxy front-end design and compare its performance against the original design. In chapter 5, we introduce our new null-message algorithm and evaluate its performance. In the last chapter, we present concludes and discuss the possible future works.

CHAPTER 2

ORIGIN AND HISTORY OF THE PROBLEM

2.1 Parallel Discrete Event Simulation

The Discrete Event Simulation (DES) models the systems as a group of components and the operations of the systems as a sequence of events. Three main parts determine the correctness of the DES simulation. These include the state of components, the event list, and the clock that represents the simulation time. During the DES simulation, each component generates, sends, receives and processes the events based on the timestamp. While handling the events, the components update their current local time to the timestamp contained in the event, and change their state according to the event type. The simulation stops when it reaches the predefined limits, such as time or number of events.

In DES programs the components run independently from each other, and communicate only by sending and receiving the events. This feature leads to the potential to parallelize the simulation, and such potential has been utilized in Parallel Discrete Event Simulation (PDES). In PDES programs, the entire simulation is partitioned into multiple Logic-Processes (LP). Each LP runs one or more simulation components and can have its own hardware resources. The events are generated and processed in a distributed manner across LP. LPs maintain their own local event list and local simulation time. The global order of events must be ensured during simulation. To ensure the global order, LPs exchange message with timestamps. Generally, synchronization algorithms provide the methods and rules for LPs of PDES programs to exchange the messages, responses for synchronizing the overall simulation, and guarantee correctness of event processing.

2.2 Synchronization Algorithms

The synchronization algorithms play key roles in PDES. It synchronizes the local simulation time at each LP and controls the overall progress of simulation. The

performance and efficiency of PDES programs depend heavily on the synchronization algorithms. There are two fundamental categories of synchronization algorithms, which include the conservative time algorithms and the optimistic algorithms.

2.2.1 Optimistic Time Synchronization Algorithm

In PDES with optimistic time synchronization algorithms, violation of timestamps during event processing is possible. Due to the fact that each LP can run at different simulation time, events with earlier timestamps can be received later than those with later timestamps at some LPs. If the events with later timestamps have not been processed, the synchronization algorithm typically reorder the event list and allow the events with earlier timestamp be processed first. However, if the events with later timestamps have already been processed and committed, violation occurs and the LPs must handle the violation accordingly to guarantee the correctness of overall simulation.

Generally, the process to handle violation includes two steps. At first, the LPs nullify the effect of violating events by rolling back the state of its components. Then, the LPs process the events with earlier timestamp, change the states of components, and continue to process the remaining events. Depend on design and implementation of the synchronization algorithms, LPs might reprocess the events that cause the violation after the violation handling.

The violation recovery progress for optimistic time synchronization algorithm is most commonly implemented with Time Wrap [9]. In Time Wrap, each LP saves the states for all or a portion of components before processing an event. It discards the changes and rolls back to the saved states if violation occurs. Saving the states of the simulation components can be memory consuming and limit the efficiency and scalability of the simulation. To address this issue, researchers proposed Global Virtual Time (GVT) [10] and Reverse Computation [11]. For optimistic synchronization algorithms use GVT, each LP maintains a record for local earliest timestamp of unprocessed event, and the timestamps are gathered to calculate the global earliest timestamp of unprocessed event. The GVT is set as the global earliest timestamp of unprocessed event. The algorithms use the GVT as a threshold such that all saved states with timestamps earlier than the

threshold can be safely deleted to reduce the memory consumption during simulation. However, some LPs can have local simulation time far ahead the GVT. Therefore, in some cases, the memory usage for saving the states in GVT based parallel simulation can still be high.

Reverse Computation is an alternative approach to mitigate the memory usage for optimistic time synchronization algorithms. In Reverse Computation based algorithms, additional codes called inverse codes are required to handle the roll back. The inverse codes response for nullify the effect of violation events. In most case, the inverse code must handle different types of events separately. So, each type of event should have its own inverse handler or equivalents. The Reverse Computation based algorithm saved the memory usage efficiently, but the inverse codes introduce considerable complexity to design and implementation that most commonly must be handled manually.

2.2.1 Conservative Time Synchronization algorithm

In the conservative time synchronization algorithms, LPs process events only when no violation is guaranteed. So, the roll back mechanism is not necessary. However, the synchronization algorithms need to determine when it is safe to process an event during the simulation. The knowledge that each LP has alone is not enough to calculate the safe timestamps. It requires additional information from other LPs. In most cases, the additional information is a lower bound for timestamps that another LP will not generate any earlier event. There are two broad categories of conservative synchronization algorithms, which include asynchronous algorithms and synchronous algorithms. They differ primarily in how to calculate the lower bound of timestamps.

The synchronous algorithms calculate Lowest Bound TimeStamp (LBTS) and store the LBTS on every LP. After the LBTS is ready, it acts as the no violation threshold at each LP. Researchers proposed several different algorithms to calculate the LBTS including [12], [13], [14] and [15]. The actual procedures for these algorithms are differing from each other, but the underlying processes are similar to GVT calculation in the optimistic time synchronization algorithms. In PDES with synchronous algorithms, a transient message is a message that is still in transfer during LBTS calculation. The

transient messages with earlier timestamps than the LBTS can be received after the LBTS calculation and cause problem. It is important for synchronous algorithms to handle the transient message correctly. [12], [13], [14] and [15] handle the transient messages in different manners and provide working solutions to this issue.

For the asynchronous algorithms, LPs exchange messages that contain no violation timestamps with other LPs, and calculate local no violation timestamps according to the received messages. No global synchronization is required for asynchronous algorithms. LPs run heterogeneously and can have different local simulation time. Chandy-Misra-Bryant (CMB) algorithm [16][17] is a well-known asynchronous algorithm. In PDES with CMB algorithm, each LP has the knowledge of the minimum simulation time for an event to propagate to any neighbor. In the original CMB algorithm, each LP exchanges messages with all neighbor LPs that contain timestamps equal to minimum timestamp of local unprocessed events plus the lookahead which equals to event propagating delay to that neighbor. This message is called a null-message. When a certain LP receives all the null-messages from neighbors, it chooses the lowest timestamp among null-messages and set the timestamp as no violation threshold. After the threshold is set, LP processes any event with an earlier timestamp than the threshold.

In the original CMB algorithm, deadlock is possible when two or more neighbor LPs have zero lookahead to each other. When zero lookahead LPs try to process events with the same timestamps simultaneously, the timestamps in null-messages are set to the events' timestamps. Therefore, very LP is not able to process the event and needs to wait for each other to move the timestamps in null-messages forward, while no one can escape from the waiting. To resolve the deadlock problem, techniques such as deadlock detection and recovery [18] and simulation time window [19] have been proposed by researchers.

Another important drawback with the original CMB algorithm is the number of null-messages that be generated during the simulation. In the original CMB algorithm, each LP sends null-messages to all known neighbors after processing every event. The number of null-messages grows exponentially with system scale and severely limits the performance and scalability. To mitigate this problem, Misra proposed a request and

response-based null-message algorithm in [20]. In Misra's approach, LPs send out requests to neighbors only when no more local events can be safely processed and neighbors response to the request with null-messages accordingly.

2.3 Maximizing Lookahead for Null-message Based Algorithms

In null-message based PDES program the Lookahead is a prediction for the amount of simulation time that a given LP will not generate any event. Generally, with a given simulation models and hardware resources, improving the lookahead leads to better performance. Low or zero lookahead null-message based PDES programs can perform equal to or even worse than the sequential simulation due to the networking, message processing and other types of overheads. Therefore, improving lookahead is critical for achieving reasonable performance for null-message based PDES.

The lookahead in null-message-based parallel simulation of multi-core systems relates closely to the precision of simulation. In low precision simulation, LPs generate events in scale of hundreds of clock cycles or even more. Because of the low event rate it can have large lookahead between components. However, in high precision simulation such as cycle-by-cycle precision simulation, obtaining reasonable lookahead is challenging due to two facts. On one hand, every component can generate events that affect other LPs at any clock cycle. On the other hand, the events propagating delays between components are typically only a few or even one clock cycle. These two facts result in very low lookahead when traditional application independent null-message optimization is employed. To improve the lookahead, find other source of lookahead than the event propagating delay is necessary.

In parallel simulation of multi-core systems, the domain-specific knowledge is the additional information that LPs can acquire from the components that are assigned to it. With the domain-specific knowledge, a LP can potentially increase the lookahead by utilizing the components' processing delay and (or) delays between components inside that LP. From our perspective, these types of delays are valuable sources of lookahead to improve the high precision parallel simulation of multi-core systems.

2.4 Partitioning Schemes

With a given PDES program and parallel hardware, the partitioning scheme can have significant effect on performance and parallel efficiency to the simulation. It plays an important role in both optimistic and conservative parallel simulation. Well-designed partitioning schemes can have great improvement over non-optimized schemes [21]. Because of its importance, partitioning schemes for a general network has been studied extensively by researchers, especially for parallel simulation of Very Large Scale Systems. In such simulation, it can have components in the scale of hundreds of thousands or even more. The complexity in partitioning such systems is high, and manual partition is time consuming. So, researchers developed automatic partitioning tools to partition such systems.

A commonly used automatic partitioning tool is hMETIS [21], which was designed and implemented by Karypis and colleagues. The experimental results of hMETIS indicated that it achieved 9%-30% improvement of performance against other partitioning schemes they evaluated. Their results confirmed the effect that partitioning schemes can have on parallel simulation for very large scale systems. However, their work focused mainly on Large Scale Integrated (VLSI) systems which differ from the multi-core systems we are interested in both system operation and the complexity of systems.

To our knowledge, little study has been conducted for partitioning the null-message based parallel simulation for cycle-by-cycle precision multi-core system. And there is no existing general conclusion or guide that could be applied to partition such systems. Therefore, we measured the effect of partitioning schemes can have on parallel simulation of multi-core system in the chapter 3.

2.5 Parallel Simulation of Multi-core Systems

The parallel simulators for multi-core systems have been developed by the computer architecture community to simulate the complex modern multi-core systems during last two decades. The existing parallel simulation systems employed different types of

synchronization algorithms. And, most of them utilized the LP level parallelism to explore benefit of parallel hardware.

Reinhardt and colleagues proposed the Wisconsin Wind Tunnel (WWT) [22] at 1993, which is an early parallel simulation infrastructure for simulating parallel shared-memory architectures. The WWT acts like a virtual machine, it executes the instructions of target architectures [22], directly on the host machine [22] that runs the simulation. It employs the quantum-based synchronization algorithm. After every quantum of *Q* clock cycles of the target architecture, the program performs a global synchronization. The violation of timestamp is possible for a quantum-based synchronization algorithm. The WWT doesn't model the interconnection network. The latency of inter-node messages is fixed at *T* cycles of target architecture. If assign each node of target architecture to an independent LP and the quantum $Q >$ latency T， no violation is guaranteed.

The Structural Simulation Toolkit (SST) [4] is a parallel simulator developed by Sandia National Laboratories. The SST is designed for simulation of new technologies in the field of supercomputing. It uses a barrier-based conservative synchronization algorithm in parallel simulation. During the simulation *sync objects* are created in a certain periodic of time, and when such objects be processed the simulator's kernel perform a global synchronization.

The SlackSim [6] is a parallel simulation system for computer architecture that utilizes an optimistic time synchronization algorithm. In the parallel simulation of SlackSim, it utilizes a GVT-based mechanism and a quantity called *slack* to determine how far each LP can move before next global synchronization. The GVT in SlackSim is the smallest local simulation time among LPs. During the simulation LPs can at most process the events with timestamp up to GVT + *slack* before the global synchronization. When the global synchronization occurs, the GVT moves such that the upper bound of timestamp also moves. The fundamental process is similar to quantum-based simulation, but the difference is that quantum-based algorithms use explicit barrier synchronization.

Chidester and George [23] performed a design evaluation for parallel simulation of chip-multiprocessors and examined a few synchronization algorithms as the design choice. The algorithms they evaluated include both null-message-based and barrier-based algorithms. In their work, they used a non-optimized null-message-based algorithm that

achieved a speedup of 2 against traditional sequential simulation. In our knowledge, there is no existing parallel simulation system designed for multi-core systems implemented with optimized null-message based synchronization algorithm.

2.6 The Manifold Project

The Manifold Project is an open source parallel simulation infrastructure that designed to simulate future generation multi-core and many-core computer architectures. To achieve the goal of easy implementation and integration of new components, it adopts component-based design and standardized interface among the components. To effectively simulate the multi-core and many-core systems with high complex, Manifold employs the PDES approach.

A typical system that Manifold simulates can be seen in Figure 1. As we can see, the system model has a certain number of cores, caches, memory controllers and an interconnection network. Each core has and connects to its own caches, and the caches connect to the interconnection network. The system can have one or more memory controllers that are independent from both the core and the cache and are also connected to the interconnection network. The interconnection network consists by a certain number of routers that connect to each other. Each cache and memory controller is actually connected to one router inside the interconnection network. When performing the parallel simulation, the program is partitioned into multiple LPs. Each LP can operate one or more above mentioned architecture components, and the components can be different types. Each LP runs the Manifold kernel that responsible for synchronizing the overall simulation. Currently, the Manifold project supports both LBTS and null-message-based conservative time synchronization algorithms, and quantum-based simulation that allow a small portion of inaccuracy but improved simulation speed.

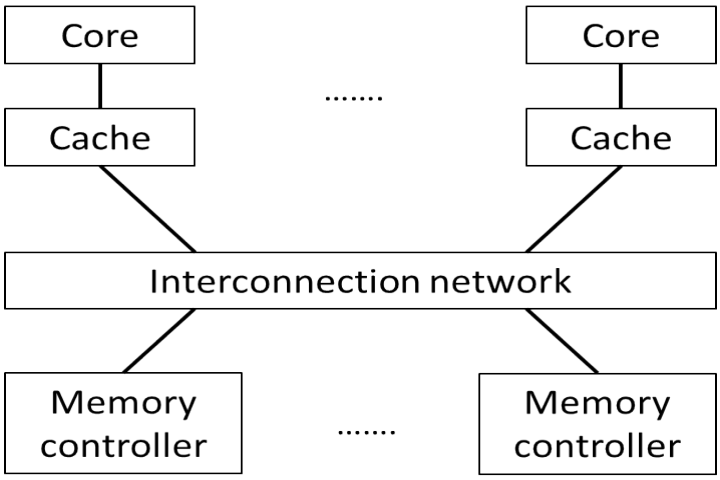


Figure 1: Typical system model of Manifold

CHAPTER 3

THE EFFECT OF PARTITIONING ON SIMULATION OF MULTI-CORE SYSTEM WITH NULL-MESSAGE BASED ALGORITHM

The Current Manifold Project does not use any automatic partitioning tool. Because the scale of the multi-core systems are relatively small compare to Very Large Scale Integrated systems that generally need assistance of automatic partitioning tools. In our study, we compare three different manual partitioning schemes, examine their effects on performance and parallel efficiency, and explain the reasons for the effects.

3.1 Schemes We Tested

The multi-core systems we tested have a star topology that consisted by an interconnection network and a number of nodes connected to the network. Each node contains either a memory controller or a core and its cache slices. The inter-component links that can be partitioned are falling into one of the following categories:

- Core-cache links
- Cache-network links
- Memory controller-network links
- Router-Router links

In partitioning schemes, cut a link means that assign the component on different sides of the link to different LPs. I our tests, we didn't cut the core-cache link because the cache hit rate is well about 95% for all benchmarks. The high hit rate means that if we cut this type of link it would introduce at least 20 times more inter-LP events than cutting the cache-network link. Therefore, we always cut the cache-network link rather than the core-cache link.

The memory controller is a very simple component that would not require much processing power. However, it exchanges a considerable number of packages with the interconnection work. If we cut this type of link, it brings more null-messages to the

simulation and could hurt the performance. We didn't cut the memory controller-network links, and simply assigned each memory controller to the same LP as the router it connected to.

The interconnection network acts as a hub in multi-core systems, all caches and memory controllers communicate with each other and achieve the coherence via the interconnection network. With the scale of the system goes up, the work load grows at the network LP. On one hand, it needs more routers to allow all the components interact and generates more events. The table 1, shows the average number of events that a core-cache LP process and that processed by the network LP during a 10 million cycles simulation. As we could see, when the system scale reached 16-core level or above, the events generated on the network LP exceed that of a core-caches LP, and it increases sharply when the system scale grows. On the other hand, the numbers of cache-network links that the network LP has also increase with system scale and results in more null-messages at the network LP. Handling the null-messages can further increase the work load of network LP. If we don't cut the router-router links, the high work load on the network LP can potentially be the bottleneck and slows down the entire simulation.

Table 1: The number of events of core-cache LP and network LP

Sys model	Core-cache events	Network events
16-core	16019517	21872648
32-core	7667381	57477376
64-core	13515153	98942717
128-core	11367344	152586095

However, if the partitioning schemes cut the router-router links, it can generate more null-messages and synchronization overhead compare to cutting only cache-network links, which might have negative effects on the performance. So, the effect of cut router-router links remains unclear, and the partitioning schemes we tested are differ only in the how the interconnection network is cut.

Scheme 1: The first partitioning scheme is called 1-part. The partitioning scheme can be seen in Figure 2. In this scheme, the entire interconnection network and all the memory controllers are assigned to a single LP. It requires the number of core + 1 LPs to run the simulation.

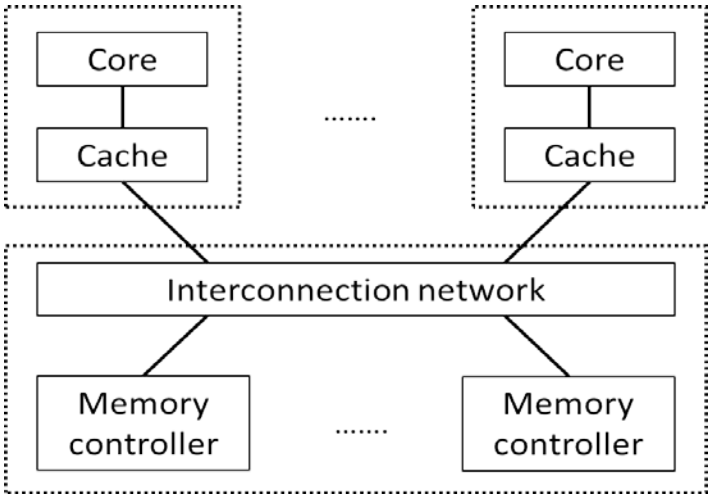


Figure 2: 1-part

Scheme 2: in the 2-part scheme, the interconnection network is divided into two equal half, and each half is assigned to a separate LP. It can be seen in Figure 3. In simulation with a 2-part scheme, the number of core + 2 LPs is needed.

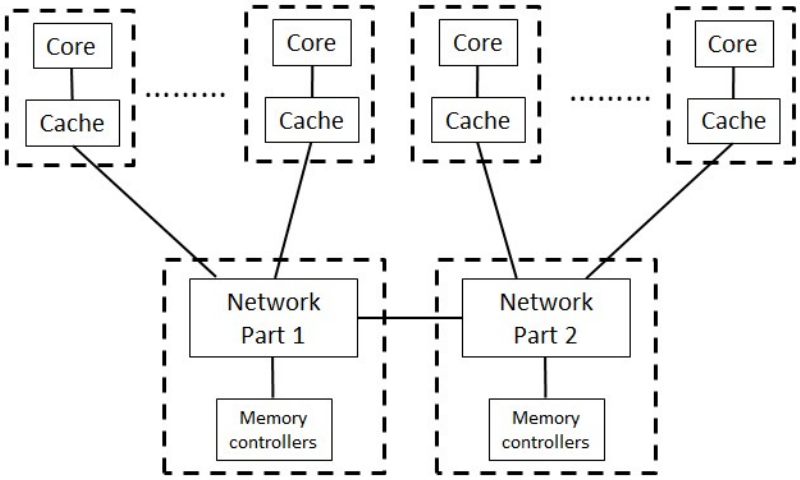


Figure 3: 2-part

Scheme 3: Y-part, as shown in Figure 4, this scheme divides the network into Y parts, where Y is the Y dimension in an X×Y torus network. Each row of routers in the interconnection network and all the memory controllers that connect to these routers are

assigned to an independent LP. In this scheme, the number of cores + Y LPs in total is necessary.

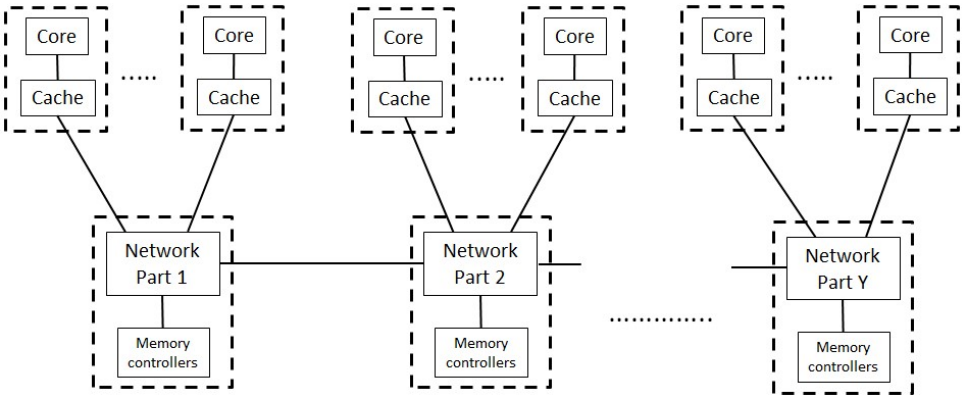


Figure 4: Y-part

3.2 Design of Experiments

The system model we built and tested includes 16, 32, 64 and 128 cores multi-core systems. All four system model have the same architecture. Each core has its own private L1 cache and a shared L2 cache. One memory controller is attached to the system for every eight cores. All the memory controllers and caches are connected to an interconnection network Torus topology. The Torus network is basically a mesh network that has edge nodes connected to each other. Figure 5 shows a 4×5 Torus network.

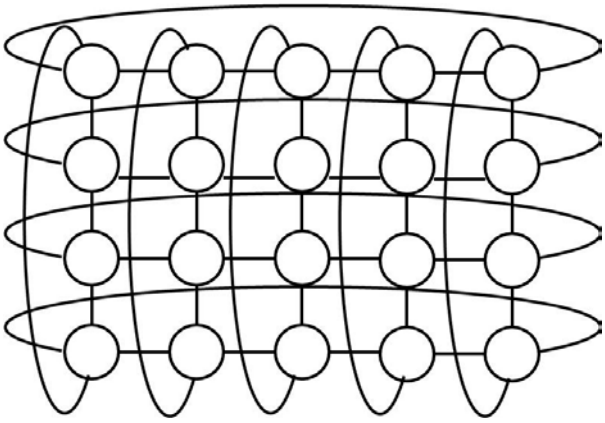


Figure 5: 4×5 Torus network

For 16-, 32-, 64- and 128-core systems, 4×5, 6×6, 9×8 and 12×12 Torus networks are applied, respectively. The core model we used is the x86 cycle-by-cycle accuracy model called Zesto [42]. Both L1 and L2 caches employ the write-back [24] mechanism and the shared L2 caches implement Modified-Exclusive-Shared-Invalid (MESI) [25] coherence protocol. All systems run the same credit-based flow control protocol [43] alone the core-cache-network path and among routers inside the network. For simplicity reason, all components are registered to the same clock and running at the same frequency.

Our experiments were conducted on a Linux cluster which has eight nodes that are connected by an underlying interconnection network. Each node runs dual Intel Xeon X5670 processers, which has six cores and two physical threads per core with the hyper-threading technology enabled. So, there are 24 usable physical threads available per node. The operation system of the cluster is RHEL release 6.3 [39] with Open MPI version 1.5.4. [40] In all tests, we provided enough hardware resources so each LP could run on an independent hardware thread.

In tests of 1-part scheme, for 16-core system model, we used only one node to run all 17 LPs. For 32-core system model, we used two nodes in the cluster with assigning the network LP and half of the core-cache LPs to one node, while the other half of core-caches LPs to another node. Similarly, we divided the program into three pieces in the tests of 64-core system model with 1-part scheme. The 64 core-cache LPs were separated into three roughly even pieces, which have 22, 21 and 21 LPs respectively. We assigned each piece to an independent node, and put the network LP at the node that runs 22 core-cache LPs. For 128-core system model with 1-part 129 LPs were divided into six roughly equal pieces and distributed across six nodes. Therefore, one node runs 22 core-cache LPs together with network LP, one node run 22 core-cache LPs, while each of the remaining four nodes runs 21 core-cache LPs.

The LP assignments for different system models of 2-part are very similar to that of 1-part. The core-cache LPs were divided and assigned in the same manner as 1-part, and the only difference is that there were two network LPs in the tests for 2-part scheme. The two network LPs are always assigned to the same node to reduce cross machine

router to router messages that typically have high delay, and these two LPs are assigned to the same node as the tests of 1-part scheme.

In tests for Y-part scheme, to eliminate the cross machine router to router messages, we always assigned all network LPs to the same node. And, in the tests for 16- and 32-core system models, we applied the same LP assigning strategies with the other two schemes. For tests of 64-core system model, network LPs requires eight hardware threads in total such that three nodes are just enough to hold all sixty four core-cache and eight network LPs. So, we put sixteen core-cache LPs and eight network LPs into the same nodes, and twenty four core-cache LPs to each of the rest two nodes. For 128-core system model, it has 12 network LPs and 128 core-cache LPs, We assigned 12 network LPs and 12 core-cache LPs to one node, and again divided the remaining 116 core-cache LPs roughly into five equal pieces and assigned them to five other nodes on the cluster.

In our tests, the total time consumption (*t_total)* and the time consumed by each LP to gather and process null-message (*t_gp)* were recorded. For *t_total* we used the *time* Linux command to record it. For the *t_gp* we inserted two function call of *clock_gettime()* into our null-message gather and process function, one at the beginning to record down *t_start*, one at the end to record down *t_end*. We measured the execution time of each call of null-message gather and process function by *t_gp = t_end – t_start*, and we accumulated the *t_gp* after each call and generated the total *t_gp* at the end of simulation. We performed tests with five randomly chosen PARSEC benchmarks [26] for every combination of system model and partitioning scheme. The benchmarks included: *facesim, ferret, freqmine, streamcluster* and *vips.* Each test was repeated for three runs because the traffic on the interconnection network of the cluster can vary during program execution and brings uncertainty to the test results. The final results presented in the following section are the average value over three runs.

3.3 Evaluation

Our experiments show clearly the effect of the partitioning schemes. The best partitioning scheme y-part consistently outperformed the other two and achieved about three times speedup against the worst case, in the best case of 128-core system model tests. Though it

requires more hardware resources than the other two partitioning schemes, Y-part also achieves better parallel efficiency when the system scale reached 32-core and above. We found the major reason for the difference in performance and parallel efficiency can be traced to the time that network LPs consumed to gather and process the null-messages. With increasing system scale the network LPs in 1-part and 2-part spends growing time to handle the null-messages, and the null-message handling time becomes the major part of the total simulation time.

3.3.1 Total Time Consumption to Run the Simulation and Relative Speedup

Table 2, 3, 4 and 5 shows the total time consumption to run the simulation and speedup against the sequential simulation for 16-, 32-, 64- and 128-core systems, respectively. As shown in the Table 2, in tests for 16-core system model, the differences of performance for three partitioning schemes are relatively small. 1-part and 2-part have almost identical performance, while Y-part has from 2% to 9% better performance compared to them. All the schemes have from 4.9X to 5.9X time speedup against the sequential simulation.

Table 2: Simulation time in seconds and relative speedup for 16-core model

Benchmark	Sequential	1-part	2-part	Y-part
facesim	3996	735 (5.4X)	734 (5.4X)	696 (5.7X)
ferret	4021	818 (4.9X)	820 (4.9X)	750 (5.4X)
freqmine	4155	756 (5.5X)	750 (5.5X)	707 (5.9X)
streamcluster	4089	739 (5.5X)	739 (5.5X)	718 (5.7X)
vips	4116	720 (5.7X)	730 (5.6X)	708 (5.8X)

Table 3 presents the test results for 32-core system model. As we can see from this table the performance gap increases when compared to the test results of 16-core system model. Y-part achieves at least 7% and 14% less total time consumption against

2-part and 1-part, respectively. In test for ferret benchmark, Y-part outperformed 1-part by 40%, which is a great improvement compared to the 9% it has in tests for the 16-core model. On the other hand, different from tests for the 16-core model, in tests for the 32-core model, 2-part always has from 3% to 9% better performance than 1-part. And, 1-part always has the worst performance among three schemes. When compared to sequential simulation, each scheme has better speedup compared to the tests for the 16-core model. 1-part has from 5.6X to 7.3X speedup, 2-part has from 6.3X to 7.7X speedup, and the best case Y-part has from 7.9X to 8.4X speedup.

Table 3: Simulation time in seconds and relative speedup for 32-core model

Benchmark	Sequential	1-part	2-part	Y-part
facesim	9023	1241 (7.3X)	1192 (7.6X)	1073 (8.4X)
ferret	8600	1531 (5.6X)	1368 (6.3X)	1087 (7.9X)
freqmine	8555	1241 (6.9X)	1208 (7.1X)	1085 (7.9X)
streamcluster	8748	1210 (7.2X)	1180 (7.4X)	1044 (8.4X)
vips	8634	1230 (7.0X)	1121 (7.7X)	1046 (8.2X)

As we predicted, the differences in total time consumption and speedup further increased in tests for the 64-core system model. Y-part also consistently achieved best performance for every benchmark, and it has form 7.8X to 11.4X speedup compare to sequential simulation. Y-part has better speedup in four out of five benchmarks than it has in tests for 32-core model. The only exception is that in tests for *streamcluster* benchmark in tests for 64-core model, Y-part has 7.8X speedup which is less than the 8.4X it achieved in the same benchmark with 32-core model. On the contrast, the speedup of 1-part and 2-part mostly decreases in tests for the 64-core model when compared to the tests for 32-core model. For 1-part, it has only from 3.7X to 5.1X speedup compared to sequential simulation, which decreased from 9% to 49%. Similarly, 2-part only has better speedup compared to tests in 32-core model in the *ferret* benchmark. For the rest of benchmarks its speedup decreased from 8% to 32%. Additionally, due to the fact that more hardware resources are used in tests for 64-core

systems model but lower speedup is achieved, 1-part and 2-part shows worse scalability than y-part.

Table 4: Simulation time in seconds and relative speedup for 64-core model

Benchmark	Sequential	1-part	2-part	Y-part
facesim	18163	3740 (4.9X)	2656 (6.8X)	1592 (11.4X)
ferret	18561	3632 (5.1X)	2662 (7.0X)	1716 (10.8X)
freqmine	17959	3753 (4.8X)	2753 (6.5X)	1956 (9.2X)
streamcluster	14079	3786 (3.7X)	2796 (5.0X)	1813 (7.8X)
vips	18000	3619 (5.0X)	2717 (6.6X)	2080 (8.7X)

The table 5 shows the tests results for the 128-core system model. The Y-part clearly achieved best performance among three schemes we tested. It has from 14.7X to 17.7X speedup compared to sequential simulation, which increased from 31% to 127% compared to the tests for the 64-core system model. However, 1-part and 2-part again failed to improve the speedup with additional hardware resources in most of test cases. 1-part and 2-part only has better speedup in *streamcluster* benchmark compared to the tests for the 64-core systems, while 2-part has very close speedup in tests for *vips* benchmark. We believe with the system scale increasing the trend would persist, where Y-part has increasing speedup, and the other two schemes would fail to do so.

Table 5: Simulation time in seconds and relative speedup for 128-core model

Benchmark	Sequential	1-part	2-part	Y-part
facesim	30295	7258 (4.2X)	4570 (6.6X)	2029 (14.9X)
ferret	30410	8455 (3.6X)	4864 (6.2X)	2062 (14.7X)
freqmine	29604	7517 (3.9X)	5551 (5.3X)	1936 (15.9X)
streamcluster	36300	7281 (5.0X)	5335 (6.8X)	2054 (17.7X)
vips	32413	7396 (4.4X)	4858 (6.7X)	2076 (15.6X)

3.3.2 Parallel Efficiency

We compare the parallel efficiency of the three partitioning schemes in a normalized manner with the following equation:

$$Normalized\ Efficiency = \frac{\frac{t\_total\_1part}{t\_total}}{\frac{num\_lp}{num\_lp\_1part}} \quad \textbf{(1)}$$

In the above equation *t_total_1part* is the total simulation run time for 1-part scheme, *t_total* is the total simulation run time for the targeting scheme, *num_lp_1part* is the total number of LPs that 1-part uses, and *num_lp* is the total number of LPs that targeting schemes uses. Follow this equation the parallel efficiency of 1-part is normalized to one, and if the normalized efficiency of the targeting scheme is greater than one, it means that the targeting scheme has better parallel efficiency that 1-part.

Figure 6 shows the normalized parallel efficiency that averaged over five benchmarks. As we can see, in tests for 16-core system model, 1-part has the best efficiency, while Y-part has the worst. This is due to the fact that all three schemes have very similar performance, while 1-part used less hardware resources than the other two schemes. However, with growing system scale, Y-part gradually outperforms the other two schemes. When the system scale reaches 32-core, Y-part has slightly better parallel efficiency than the other two schemes. And, in tests of the 64-core system model, Y-part outperforms 1-part by about 80% and 2-part by about 40% in terms of parallel efficiency. For the 128-core system, the gap of parallel efficiency grows sharply. Y-part achieves 244% and 196% better parallel efficiency against 1-part and 2-part, respectively. Based on the observed results we believe good partitioning schemes also improve the parallel efficiency.

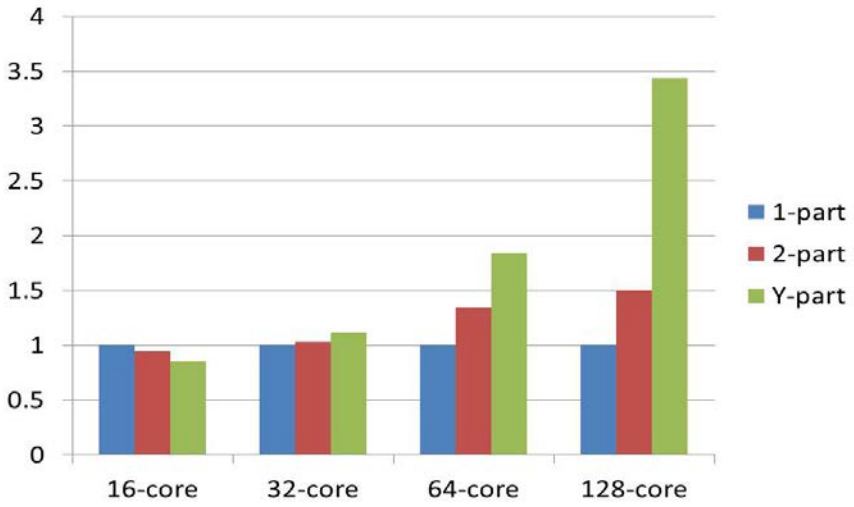


Figure 6: Normalized parallel efficiency

3.3.3 Time to Gather and Process the Null-message

Tables 6, 7, 8 and 9 show the null-message gather and process time (*t_gp*) and the percentage of *t_gp* against the total time consumption. As we can see from the tables, the *t_gp* increases dramatically with growing system scale for 1-part and 2-part. 1-part consumes from 118 to 185 seconds to handle the null-messages in tests for 16-core model, and these numbers increased to 3365-4096 seconds when the system scale reached 128-core. Additionally, the percentage of *t_gp* grew from roughly 20% in tests for 16-core model to 50% in tests for 128-core model. The situation is similar in tests for 2-part. The *t_gp* grew from 117-151 seconds to 1876-1990 seconds with system scale increases. Also, the percentage increased from roughly 20% to 40% against the total time consumption.

On the contrast to 1-part and 2-part, Y-part had much less null-message gather and process time. In tests for 16-core model it spent from 113 to 136 seconds to handle the null-messages, which is roughly the same as the other two schemes. However, when system scale reached 32-core and above, Y-part consumed far less time to handle the null-message. In tests for 32-core system model, *t_gp* Y-part is from 118 to 165 seconds which is about 60% of *t_gp* of 1-part and 2-part. For 64-core system model, 1-part and 2-

part consumed about 5 times and 2.5 times more time to handle the null-messages than Y-part. In tests for 128 core systems, Y-part spent from 385 to 446 seconds in null-message gathering and processing, which is about 8 times and 4 times less than 1-part and 2-part, respectively. Additionally, the percentage of *t_gp* against the total time consumption remains at about 20% regardless of the system scale in tests of Y-part, and we believe the trend will be kept with the system scale grows.

Based on the above observation, in simulation of multi-core system, if we do not partition the network LP, when the system scale grows the network LP has increasing number of cache-network links. Increasing number of cache-network links results in more null-messages and more null-message gather and process time during simulation at network LP. Therefore, the synchronization overhead at network LP grows with system scale and finally becomes the major impediment for performance. As the result, the scalability of 1-part and 2-part are seriously limited. Partitioning the network mitigates the synchronization overhead by reducing the total number of null-messages per network LP. Because of the effect of *t_gp*, Y-part consistently achieves the best performance.

Table 6: Null-message gather and process time and percentage for 16-core

Benchmark	1-part	Per. 1-pt	2-part	Per. 2-pt	Y-part	Per. Y-pt
facesim	131	17.80%	133	18.10%	132	19.00%
ferret	185	22.60%	151	18.50%	136	18.10%
freqmine	137	18.00%	146	19.40%	130	18.40%
streamcluster	123	16.60%	122	16.60%	113	15.80%
vips	118	16.40%	117	16.00%	115	16.30%

Table 7: Null-message gather and process time and percentage for 32-core

Benchmark	1-part	Per. 1-pt	2-part	Per. 2-pt	Y-part	Per. Y-pt
facesim	265	21.30%	270	22.70%	165	15.40%
ferret	426	27.80%	323	23.60%	165	15.20%

freqmine	271	21.90%	284	23.50%	127	11.70%
streamcluster	213	17.60%	258	21.90%	123	11.80%
vips	232	19.00%	232	20.70%	118	11.30%

Table 8: Null-message gather and process time and percentage for 64-core

Benchmark	1-part	Per. 1-pt	2-part	Per. 2-pt	Y-part	Per. Y-pt
facesim	1635	43.70%	822	30.90%	333	20.90%
ferret	1528	42.00%	845	31.70%	325	18.90%
freqmine	1684	44.90%	904	32.80%	315	16.10%
streamcluster	1651	43.60%	914	32.70%	328	18.10%
vips	1565	43.30%	833	30.70%	294	14.10%

Table 9: Null-message gather and process time and percentage for 128-core

Benchmark	1-part	Per. 1-pt	2-part	Per. 2-pt	Y-part	Per. Y-pt
facesim	3365	46.40%	1912	41.80%	429	21.10%
ferret	4096	48.40%	1990	40.90%	446	21.60%
freqmine	3751	50.00%	1966	35.40%	385	19.90%
streamcluster	3567	49.00%	1966	36.80%	399	19.40%
vips	3558	48.10%	1876	38.60%	425	20.50%

3.3 Discussion

This chapter presented a study for the effects of partitioning schemes have on the null-message-based parallel simulation of multi-core system. The partitioning schemes have been shown to have significant effects on the performance. With the system scale grows the effects become increasingly important. In the multi-core systems we study, the interconnection network is connected to a certain number of core-cache nodes. As the

system size grows, this number also increases. If the network is not partitioned, the synchronization overhead incurred on the network LP to process null-messages grows exponentially, which leads to great performance degradation. Partitioning the network can reduce the synchronization overhead and helps performance. Of the three partitioning schemes we study, Y-part appears to be a scalable solution, and outperforms the other two not only in simulation speed but also in parallel efficiency when the system scale reached 32-core level and above. Based on this study, we believe, in a null-message-based parallel simulation, any component whose inter-LP links increases with system scale should in principle be partitioned to achieve reasonable scalability.

CHAPTER 4

AN EFFICIENT FRONT-END FOR TIMING-DIRECTED PARALLEL SIMULATION MULTI-CORE SYSTEMS

The *timing-directed* [41] simulation is one important class of micro-architecture simulations. In *timing-directed* simulation the simulation system is separated into two independent but complementary parts, which are *functional* simulation and *timing* simulation. The *functional* simulation also referred as *front-end*, it emulates the behavior of the target architecture. *Functional* simulators include SimOS [27], Qsim [28] and AMD's SimNow® [29] are developed by researchers and industry computer architects. The modern *functional* simulation is very accurate in terms of simulating the functional behavior of target architecture and fast enough to run at the speed close to that of native execution. However, the *functional* simulation has its limitation, it mainly designed for simulating the functional behavior and typically don't have precise timing for components inside the target systems.

Due to the limitation of *functional* simulation, *timing* simulation is employed as the complementary of *functional* simulation. The *timing* simulation, or the *back-end*, is built with the simulation models of architecture components and used to evaluate the performance of target systems in terms of timing. The *timing simulation* is generally one or more order of magnitude slower than the *functional* simulation, but it precisely models the operation and transfer latency of architecture components in the target systems.

In *timing-directed* simulation for micro-architectures, *functional* simulation is responsible for generating the correct instruction flows or events streams for architecture components in *timing* simulation. While the *timing* simulation uses the instructions or events to update the states of architecture component with sending feedback that base on the states of components to *functional* simulation. The interactive feature of *timing-directed* simulation makes it suitable for simulation of complex systems such as multi-core systems. Also due to this feature the communication method between the *front-end* and the *back-end* is critical to achieve reasonable performance in *timing-directed* simulation.

4.1 Traditional Front-end with TCP/IP

The Manifold project supports both *timing-directed* and *trace-driven* simulation. The *trace-driven* simulation does not use the *functional* simulation. Instead, it uses the trace file that generated before simulation execution as the instruction input. The *timing-directed* simulation of Manifold can be decomposed into two major parts: the *functional simulation front-end* and the *timing simulation back-end*. The *front-end* of Manifold project is a QEMU-based [30] thread safe multi-core emulation library called QSim [28]. Qsim can boot a Linux OS and emulates the execution of a multi-threaded application on a number of virtual cores. The *back-end* core model sends request to QSim. The QSim emulates the execution of that certain core on the corresponding virtual core and response the core model with instructions and relative information such as virtual and physical addresses of the instructions. So, the instruction execution of each virtual core is essentially controlled by the corresponding *back-end* core model in a request and response manner.

For full-system simulation, QSsim provides a multi-threaded server to handle the requests from the *back-end*. The *back-end* models communicate with the server via TCP/IP using sockets. A high level view of the non-optimized full-system *timing-directed* simulation of Manifold can be seen in figure 7. During the simulation, a core model sends request to the server when instructions is needed, then blocks the execution for waiting the response. Once the response of instructions is received, it resumes the simulation progress. In other words, whenever a core model in the *back-end* needs instructions, TCP/IP communications are incurred and core models experiences the TCP/IP latency, which introduces considerable idle time into the simulation. Since it is hard to eliminate the TCP/IP transaction latency without redesign the entire protocol, it is highly desirable to find a method that hides the latency as much as possible from the *back-end's* perspective. We expect such a scheme could make the *back-end* core models run more efficiently and improve the overall simulation performance.

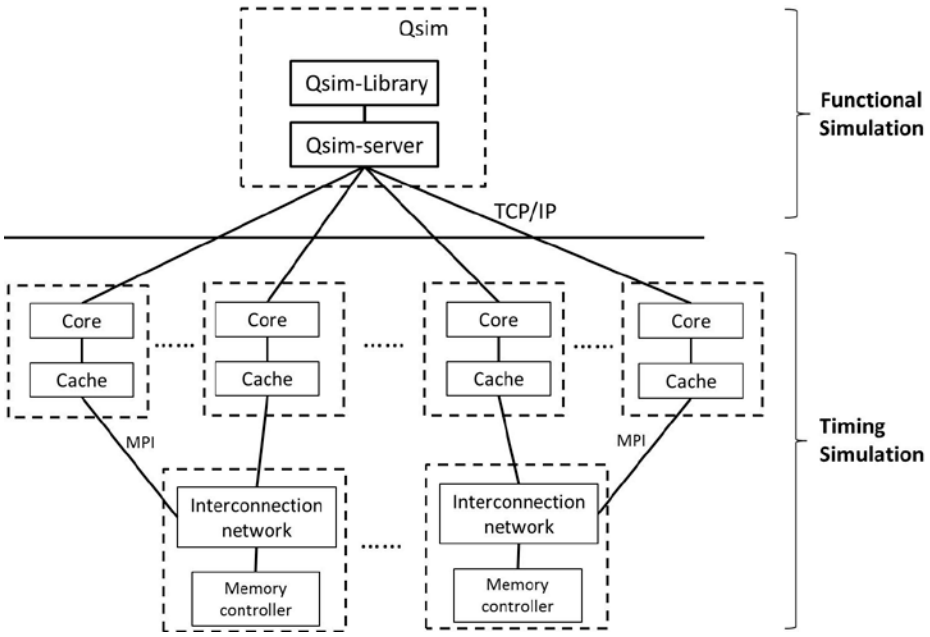


Figure 7: Manifold's *timing-directed* simulation model with client-server design

4.2 Optimizing Data Transmission with Proxy

As discussed above, we wish to hide the TCP/IP latency from the *back-end's* perspective in order to improve the performance. We attempted to achieve this goal with creating proxy processes between the server and the *back-end.* As shown in figure 8, the new design has two major differences compares to the original design, which include:

1. In the new design we introduce a number of proxy processes into the system. The proxies, together with the server, form a new and improved *front-end.*
2. The core models in *back-end* are modified. Instead of interacting with the server directly, the core models now acquire the instructions and relative information from the proxy.

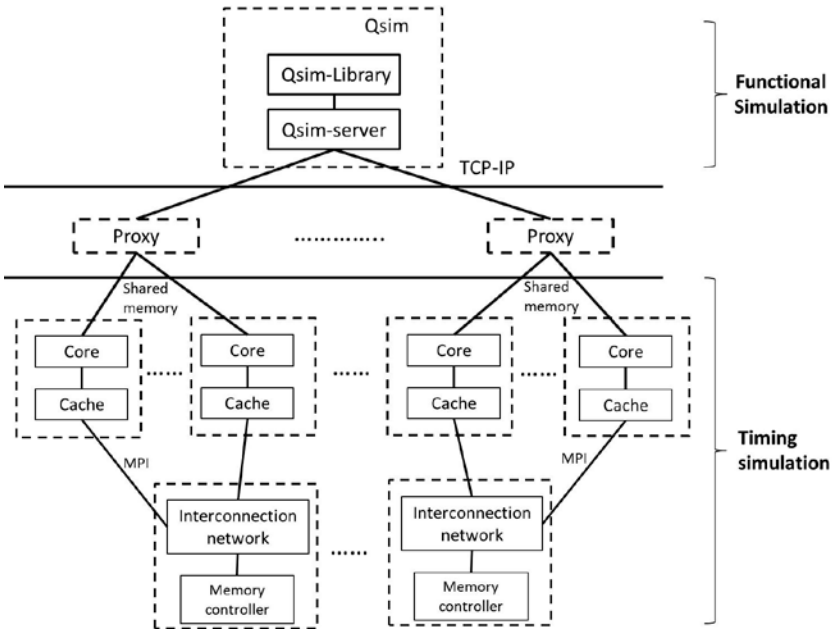


Figure 8: Manifold *timing-directed* simulation with Proxy

The proxies act as intermediary between the server and back-end. They served as the client from the QSim server's perspective and as the server from the core models' perspective. Proxies communicate with QSim server also using sockets. After the instructions are received, each proxy buffers them in an internal buffer and then copies them to shared memory segments. The core model can retrieve the instructions directly from the shared memory segments. The shared memory is a highly efficient channel for inter-process communication that allows much faster instruction fetching for the *back-end* than using the sockets. Each proxy is a multi-threaded process and serves all the core models running on the same host machine. Proxy on a given host machine allocates one thread for each core model it serves.

Each of the proxy thread acts as the producer and the core model that thread serves acts as the consumer. The proxy thread puts instructions in the shared memory segment, and the core model consumes them by removing them from the segment. This process is shown in figure 9, to simplify inter-processes synchronization, a circular buffer is employed for each of the shared memory segments. Two additional pointers are allocated, one points at the head and the other points at the tail of the circular buffer. The producer (proxy thread) moves only the tail and the consumer (core model) only modifies the head. Follow this implementation, no synchronization mechanism is needed. To

prevent the circular buffer from overflowing, the threads uses an FIFO buffer to hold instructions before put them into the circular buffer.

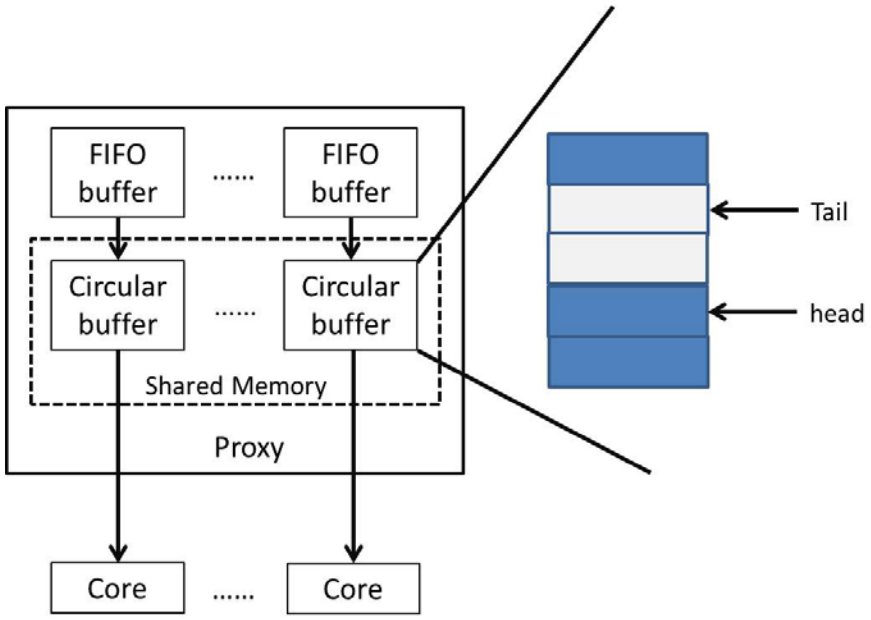


Figure 9: Implementation of Proxy

To hide the TCP/IP latency the proxy only needs to guarantees that the circular buffer is not empty. During the simulation proxy monitors the circular buffer, if the number of instructions falls below a pre-defined threshold, proxy sends request to the server to acquire more instructions. The actions of the proxy thread can be seen in follow pseudo code:

Proxy thread:

```
1:  while true do
2:      if circular buffer size bellows threshold then
3:          get instructions from server into FIFO buffer
4:      end if
5:      while FIFO not empty AND circular buffer not full do
6:          remove first item from FIFO with save it in a temporary buffer
7:      write the instruction to circular buffer
```

8: **end while**

9: sleep if did some work

10: **end while**

4.3 Design of Experiments

We built and tested 16-, 32- and 64-core system models. Each system model has a single interconnection network, a certain number of cores, caches and memory controllers. The core model we use is the Zesto, each core has its private L1 cache and a shared L2 cache, and one memory controller is created for every 8 cores. The cache models implemented with the MESI coherence protocol, and all the L2 caches and the memory controllers are connected to the interconnection network. For 16-, 32- and 64-core system 4×5, 6×6 and 9×8 Torus networks are employed, respectively. A credit-based flow control protocol is applied along the core-cache-network path, and among the routers of the interconnection network. All the architecture components of the *back-end* are registered to the same clock. For all the system models, the Y-part scheme for the interconnection network is applied for all system models.

We conducted our experiments on the same Linux cluster as what we used for the study of partitioning schemes. In all the tests, the QSim server is assigned to its own node that runs no other program. We ensure there is no other program runs at the node that we run the server by requesting all the resource on that node, and constantly monitoring the usage of that node. The *back-end* has two different types of LPs, the core-cache LPs and the network LPs. Each of the core-cache LP is assigned with two Zesto cores and their caches, which the best granularity we found during our tests. The y-part scheme that we discussed in last chapter is applied to the interconnection network.

In tests for 16-core system, one node is used to run the entire *back-end* simulation. Simulation for 32-core system models uses two nodes. All six network LPs are assigned to one node together with six core-cache LPs, and the rest of core-cache LPs are assigned to the other node. For the 64-core systems, we assigned the LPs across 3 different nodes. Eight network LPs along with 8 core-cache LPs are assigned to one node, while the rest

of core-cache LPs are divided into two equal half and each half is assigned to one of the two remaining nodes.

The size of the shared memory segments is set to 262144 (256K) bytes, which is the best value we find during our preliminary tests. Tests for each system model are performed against 6 randomly chosen benchmarks. The *vips, streamcluster* and *freqmine* benchmarks are chosen from the PARSEC, while the *barnes, cholesky* and *fmm* benchmarks are chosen from the SPLASH-2 [31]. The tests are run for 50 million cycles, and each test is repeated for 3 times to deal with the randomness that introduced by the traffic on the interconnection network of the Linux cluster. For simplicity reason, we only examine the wall-clock time of *back-end* in the following discussion, because the *front-end* is essentially controlled by the *back-end*. The simulation time of *back-end* is further decomposed into time for getting instructions, safety check, processing events, sending null messages, and receiving incoming messages.

The time for getting instructions is the cumulated time a core model spends to get instructions from QSim server or the Proxy. The safety check time is the time consumed by LPs to check whether it is safe to process next event, which is necessary in null-message-based parallel simulation. The time for processing events is the time for each LP to process events (time for getting instructions is part of this time). The time for sending null-messages is the average time spent by LPs to send null-messages. Finally, the time for receiving null-messages is the time spent by an LP to gathering incoming messages from other LPs. The total time is recorded with the Linux *time* command. The rest types of time are recorded with two calls of *clock_gettime()* function. We insert one of the call at beginning of the corresponding code segment, and the other call at the end. The execution time of the code segment is accumulated throughout the simulation.

4.4 Evaluation

The Table 10 shows the test results for 16-core system. As we can see, the Proxy design has better performance than the original design in all benchmarks we tested. The total time consumption of proxy design is from 30% to 49% less than the original design. The event processing time is similar when compare the proxy design to the original design,

which indicates that the amount of events generated during simulation is similar such that the work load of each design is almost the same. On the other hand, when changed the *front-end* from QSim server to proxy, it introduced considerable difference into other five types of time consumption. Original design consumes about 15 times more time to get the instructions. For the rest of time consumption, proxy design achieves 54% to 72%, 48% to 67% and 49% to 67% less null-message sending, safety check and processing incoming messages time compared to the original design, respectively.

Table 10: Proxy vs. Original for 16-core system

Proxy						
benchmark	Inst.	Safe	Proc.	Null	Msg.	Total
barns	9.5	607.0	1928.7	218.0	655.5	3409.2
cholesky	14.1	515.2	2148.6	181.5	569.8	3415.1
fmm	14.0	511.9	2134.3	177.3	565.9	3389.4
freqmine	13.4	506.1	2063.6	176.6	560.0	3306.2
Stream.	13.8	513.3	2121.2	176.5	568.5	3379.5
vips	12.8	533.9	2070.7	186.8	584.9	3376.4
Original						
benchmark	Inst.	Safe	Proc.	Null	Msg.	Total
barns	145.9	1180.6	1930.2	475.3	1285.7	4871.8
cholesky	275.0	1426.8	2458.0	585.2	1571.9	6042.9
fmm	307.9	1588.2	2635.2	634.9	1730.6	6588.8
freqmine	272.0	1411.8	2393.0	557.3	1543.6	5905.7
Stream.	266.5	1382.1	2390.2	574.4	1524.0	5870.7
vips	286.7	1515.4	2527.5	647.4	1690.4	6380.7

Table 11 shows the test results for 32-core system model. From the data presented in this table, we can see proxy design runs from 32% to 41% faster than original design for all the benchmarks we tested. Similar to the test results for 16-core system model, both designs have similar event processing time but huge difference in time to get instruction. The time to get instructions of proxy design is only about 5% of that of

original design. Additionally, in comparison of the time for sending null-messages, safety check, and processing incoming messages, the proxy design has from 40% to 53% less time consumption in these three execution sections than the original design. Similar event processing time with more time spent in the above three execution sections indicates that the original design spends significantly more time in waiting without making any progress.

Table 11: Proxy vs. Original for 32-core system

Proxy						
benchmark	Inst.	Safe	Proc.	Null	Msg.	Total
barns	6.4	860.5	1429.7	367.6	965.5	3623.3
cholesky	6.2	898.0	1374.0	377.6	1000.3	3649.9
fmm	6.0	862.1	1345.6	364.0	963.1	3534.9
freqmine	5.5	852.7	1307.3	363.3	948.0	3471.4
Stream.	6.1	875.3	1362.9	365.7	961.4	3565.5
vips	5.7	878.8	1339.6	370.6	968.4	3557.4
Original						
benchmark	Inst.	Safe	Proc.	Null	Msg.	Total
barns	137.5	1766.9	1628.1	776.7	2001.2	6172.9
cholesky	131.2	1492.2	1558.8	624.0	1690.0	5414.4
fmm	123.7	1479.4	1552.6	653.3	1675.7	5360.9
freqmine	119.6	1420.8	1536.6	638.1	1616.6	5212.4
Stream.	135.4	1546.4	1578.6	682.9	1755.6	5563.6
vips	119.6	1464.4	1566.2	654.9	1667.3	5352.6

In tests for the 64-core system models, the proxy design outperforms the original design by more than 45% in four out of 6 benchmarks. In tests for *freqmine* and *vips* benchmarks the proxy design also outperforms the original design for more than 30%. The time for getting instructions of original design is one order of magnitude more than that of the proxy design. The event processing time is relatively close between two designs. Proxy design has roughly 20% less time consumed in this execution section. For

the rest three types of time consumption, in tests for *barnes*, *cholesky*, *fmm* and *streamcluster*, original design spent approximately 2 times more time in execution of these sections than proxy design. And, in tests for *vips* and *freqmine* benchmarks, the baseline design has about 30% more time consumption in these three execution sections than the proxy design.

Table 12: Proxy vs. Original for 64-core system

Proxy						
benchmark	Inst.	Safe	Proc.	Null	Msg.	Total
barns	7.0	1487.4	1595.3	620.8	1638.2	5341.6
cholesky	10.6	1536.4	2053.3	624.0	1696.7	5910.4
fmm	12.4	1537.9	2283.8	618.5	1707.6	6147.7
freqmine	2.4	1528.9	1038.7	639.8	166.9	4877.3
Stream.	9.9	1508.4	1987.2	616.3	1678.0	5789.9
vips	1.5	1525.6	930.5	629.2	1641.4	4726.7
Original						
benchmark	Inst.	Safe	Proc.	Null	Msg.	Total
barns	159.1	3084.0	1979.8	1292.7	3441.5	9798.0
cholesky	236.6	3752.9	2625.7	1519.8	4159.6	12058.0
fmm	225.6	3567.2	2521.0	1467.8	3974.3	11530.2
freqmine	53.7	2321.1	1166.7	966.2	255.6	7013.6
Stream.	203.8	3520.5	2358.3	1439.9	3897.6	11216.3
vips	46.2	2209.6	1110.8	926.7	2452.7	6699.7

Table 4 compares the simulation run time of both designs to that of the sequential simulation. The numbers in the brackets are the relative speedup against the sequential simulation. From the table 4, we can see that both designs have better performance than the sequential simulation, and the speedup increases with the system scale. The proxy design consistently outperforms the original design in all test cases. It has from 42% to 104% better speedup against the sequential simulation compared to the original design, which without doubt is a significant improvement. Additionally, the speedup of proxy

design against the sequential simulation grows faster than that of original design, and it indicates that the proxy design achieves better scalability.

Table 13: Comparison with Serial Simulation

16-core			
Benchmarks	Sequential	Proxy	Baseline
barnes	15711.9	3409.1 (4.6x)	4871.8 (3.2x)
cholesky	17906.1	3415.1 (5.2x)	6042.8 (3.0x)
fmm	16594	3389.3 (4.8x)	6588.7 (2.5x)
freqmine	17811	3306.2 (5.4x)	5905.7 (3.0x)
streamcluster	17690.1	3379.5 (5.2x)	5870.7 (3.0x)
vips	18327.1	3376.4 (5.4x)	6380.7 (2.9x)
32-core			
Benchmarks	Sequential	Proxy	Baseline
barnes	26038.1	3623.2 (7.2x)	6172.9 (4.2x)
cholesky	25321	3649.9 (6.9x)	5414.4 (4.7x)
fmm	24777.1	3534.9 (7.0x)	5361 (4.6x)
freqmine	24252.3	3471.4 (7.0x)	5212.4 (4.6x)
streamcluster	23643.6	3565.5 (6.6x)	5563.6 (4.3x)
vips	24139.4	3557.4 (6.8x)	5352.6 (4.5x)
64-core			
Benchmarks	Sequential	Proxy	Baseline
barnes	60487.7	5341.6 (11.3x)	9798 (6.2x)
cholesky	64611.1	5910.4 (10.9x)	12058 (5.4x)
fmm	73299.5	6147.7 (11.9x)	11530.2 (6.4x)
freqmine	38808	4877.3 (8.0x)	7013.6 (5.5x)
streamcluster	63508.5	5789.9 (11.0x)	11216.3 (5.7x)
vips	34195.6	4726.7 (7.2x)	6699.7 (5.1x)

4.5 Discussion

As mentioned in above sections, the only difference between the two designs we discuss here is the way that the core models get the instructions, which determines the communication latency between the *front-end* and the *back-end*. Intuitively, this difference should be the major factor that leads to the performance gap. However, as shown in table 10 through 12 the communication latency is not the only factor that can

affect the overall performance, and the communication latency alone is just a fraction of the difference in total simulation time. In our experiments we observed that, when core models stall and wait for the instructions from front-end, it keeps sending and receiving null-messages and performing the safety check. Figure 10 shows the counts for core-cache LPs entering different execution sections during a 50 million cycle simulation. As shown in Figure 10, the original design with higher instruction fetch latency from the front-end entered the safety check, sending null-messages, and processing incoming messages sections significantly more times than the proxy design. The safety check, sending null-messages, and processing incoming messages sections are the synchronization overhead of the parallel simulation, by entering those sections more times the original design experiences higher synchronization overhead than the proxy design.

At the same time, the counts for entering the event processing section are identical for both designs. This section executes the codes that essentially make progresses to the overall simulation. So, by entering event processing section for the same number of times, both designs have exactly same useful work load, and it indicates that the speedup of proxy design is not coming from the reduction of useful work load.

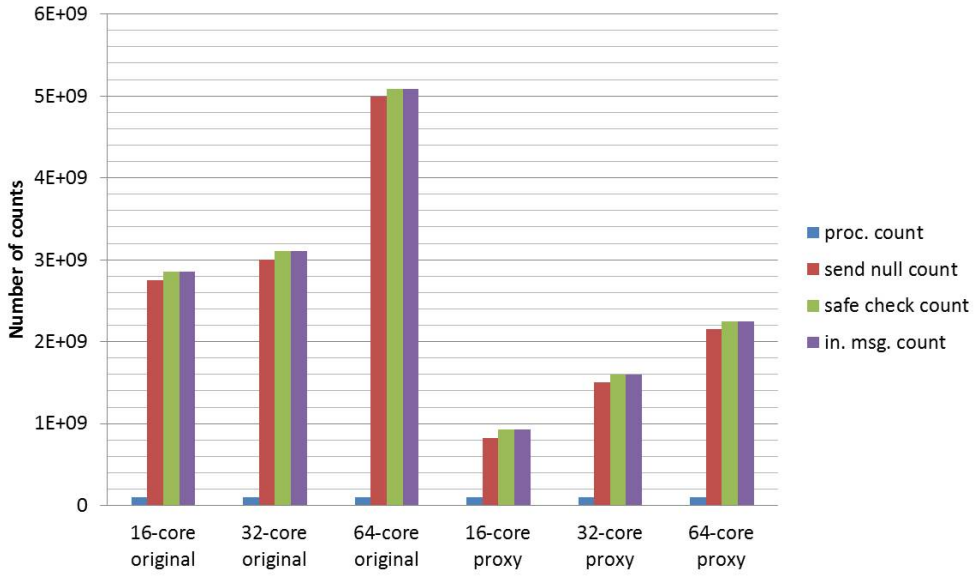


Figure 10: Counts for simulation entering different execution sections

To summarize our works, in *timing-directed* simulation of multi-core systems with full system model, the efficiency of the *back-end* to retrieve instructions from the *front-end* for its architecture components is critical for overall performance. In comparison to the original design, our new proxy design demonstrated much better overall performance. The reason for the differences in performance can be traced down to the fact that our proxy design successfully hides the TCP/IP communication latency between the *front-end* and the *back-end.* The communication latency can have a much larger effect to the parallel simulation than the effect it has alone in terms of the time consumption. Our tests results have shown that the communication latency can affect other execution sections of the parallel simulation and initiates a chain effect that essentially leads to the difference in performance. Finally, the proxy design together with the QSim server, form an efficient *front-end* with low communication overhead from the *back-end*'s perspective. We also believe our proxy design have the potential to be applied on *timing-directed* parallel simulation of other field.

CHAPTER 5

A NEW NULL-MESSAGE BASED SYNCHRONIZATION ALGORITHM

In the modern multi-core systems, the transmission latency between components such as the caches and the routers in the interconnection network is low. However, the transmission latency between components is the only source of lookahead for traditional application independent optimized null-message algorithms. Low transmission latency leads to poor lookahead, which essentially limited the performance and scalability of traditional algorithms in cycle-by-cycle precision null-message based parallel simulation of multi-core systems. To handle this issue, we propose a new application-dependent optimized null-message algorithm that utilizes the domain-specified knowledge which acquired from the components inside each LP. Our new algorithm shows significant improvement over the traditional algorithms and demonstrated good scalability on the cluster-based environment.

5.1 Traditional Null-message Algorithm

Algorithm 1 shows the implementation of the traditional application-independently optimized null-message algorithm, which we shall refer to as the baseline algorithm in the following sections. In the baseline algorithm, at the beginning each LP receives the messages with non-blocking sends and receives. Next, each LP finds out the earliest clock edge of all the clocks with recording the simulation timestamp of that edge in variable *nextClockTime*. After the earliest clock edge being recorded, LPs check all the incoming null-messages from neighbors and determine the minimum timestamp among the received null-messages and record its value in the variable *min_null*. If the *nextClockTime* is less than *min_null*, all the events scheduled for the clock edge are safe to process. Finally, each LP calculate the timestamp for null-message as min(*nextClockTime*, min_null) + Lookahead, and send null-messages out to all its

neighbor LPs if the timestamp is larger than that in last outgoing null-message. The lookahead for all links is set to a value slightly smaller than the link delay.

Algorithm 1: Baseline null-message algorithm

```
1:  while simulation not terminated do
2:      receive messages
3:      nextClock = clock whose next edge has smallest timestamp among all
        clocks
4:      nextClockTime = simulation time of next edge of nextClock
5:      receive null-messages
6:      min_null = minimum timestamp of the null-messages
7:      if nextClockTime < min_null then
8:          process all events scheduled for the clock edge
9:      end if
10:     null_ts = min(nextClockTime, min_null) + lookahead
11:     if null_ts > last null_ts then
12:         send null-messages to all neighbor with timestamp = null_ts
13: end while
```

The baseline algorithm satisfied the requests for being conservative with ensuring the events order and deadlock free. However, with only one clock cycle's inter-components delay, a core-cache LP needs to exchange a pair of null-messages with relative network LP to move the local simulation forward by one clock cycle. The case is even worse at the network LPs. It needs to exchange a pair of null-messages with each of its neighbor LPs to move forward by one cycle. Therefore, the baseline algorithm generates a huge amount of null-messages during the simulation. Table 14 shows the number of null-messages be sent and received per network-cache link at the interconnection network work side in a 10 million cycles' *trace-driven* simulation. As we

can see from Table 14, with baseline algorithm each network-cache link sends one and receives one null-message about every 0.2 clock cycle.

Table 14: Number of null-message per-link with baseline algorithm

Benchmark	Trace-driven send	Trace-driven rec
bodytrack	81,462,560	78,362,448
facesim	73,753,704	66,864,855
freqmine	67,549,584	60,451,605
streamcluster	68,659,786	61,198,045
vips	82,331,880	75,666,786

The huge amount of null-messages saturates the network of parallel hardware and leads to unreasonable performance. As shown in table 15, the baseline null-message algorithm consumes more than 24 hours to finish a 10 million cycles' simulation run for *trace-driven* simulation even with the simplest 16-core system models. When the system scale reached 32-core and above, the simulation run time kept increasing, and the high time consumption to run the simulation and hardware resource requirement can cause violation to the rules and regulations of usage of the cluster that we performed our tests on. So, we were unable to test 32-, 64- and 128-core system models with baseline algorithm and we only tested the baseline algorithm against the *trace-driven* simulation.

Table 15: Simulation time in hours to run baseline algorithm

Benchmark	Trace-driven
bodytrack	>24
facesim	>24
freqmine	>24
streamcluster	>24
vips	>24

5.2 Optimizing Null-message Algorithm

Due to Limitation of baseline algorithm we attempted to optimize the null-message-based parallel simulation of Manifold from two aspects. On one hand, we attempted to reduce the number of null-messages be sent during the simulation. On the other hand, we attempt to improve the lookahead.

5.2.1 The Send-When-Block Algorithm

The first attempt of us leads to the Send-When-Block (SWB) algorithm, which can be seen in Algorithm 2.

Algorithm 2: Send-When-Block algorithm

```
1:  while simulation not terminated do
2:      receive messages
3:      nextClock = clock whose next edge has smallest timestamp among all
        clocks
4:      nextClockTime = simulation time of next edge of nextClock
5:      receive null-messages
6:      min_null = minimum timestamp of the null-messages
7:      if nextClockTime < min_null then
8:          process all events scheduled for the clock edge
9:      else
10:         if nextClockTime is rising edge then
11:             null_ts = nextClockTime + lookahead
12:         else
13:             null_ts = next rising edge + lookahead
14:         end if
15:         if null_ts > last null_ts then
16:             send null-messages with timestamp set to null_ts
```

17: **end if**

18: **end while**

The SWB algorithm involves two optimizations against the baseline algorithm. At the first place, different from the baseline algorithm that sends out null-messages when events scheduled for a clock edge are safe to process, SWB algorithm sending the null-messages only when the LP blocked to reduce the number of null-messages generated during the simulation. At the second place, SWB algorithm slightly improved the lookahead by distinguishing the falling edge and rising edge of clock cycles. As mentioned in last section the lookahead is set to a value that close to the minimum link delay in the system, in precise words, it is set to (*min link delay – 0.1)* clock cycle. As the architecture components in Manifold simulation only send out messages at the rising clock edge, we can set the timestamp in the null-message to *timestamp of next rising edge + lookahead*, instead of min (*nextClockTime*, *min_null) + lookahead.* This idea is illustrated in Figure 11. There are four points of time: *A* is the *min_null*, or the minimum timestamp of null-messages from neighbor LPs, *B* is the current edge at which whether the events can be safely processed is evaluated, *C* is the timestamp that will be sent in null-message if the optimization is not applied, which equals to *A + lookahead*, and *D* is the improved timestamp with the optimization.

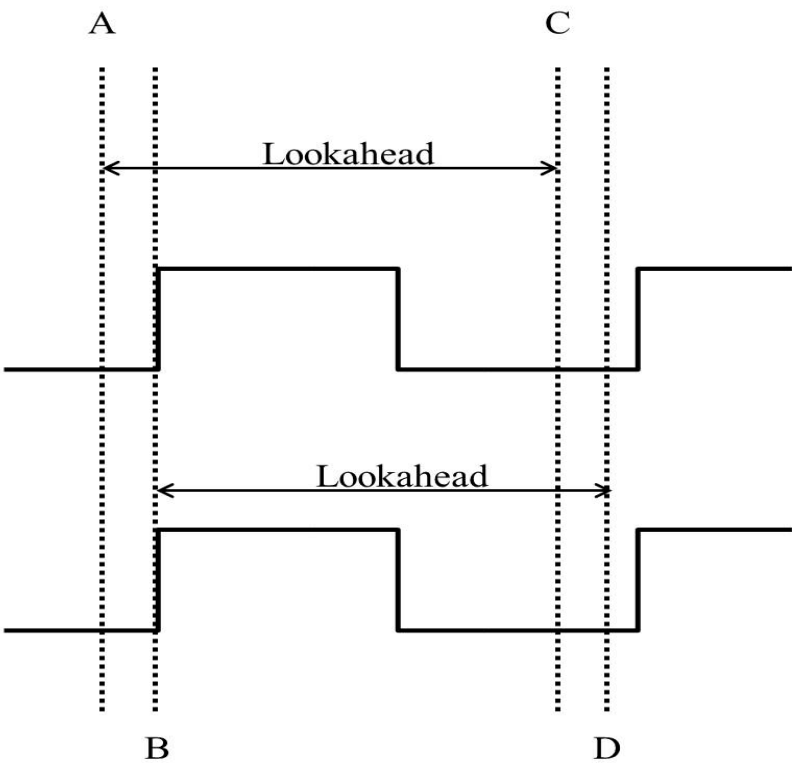


Figure 11: Lookahead improvement for SWB algorithm

With the optimization shown in Figure 11, if the *B* is a falling edge, then the point *D* can be further improved to the timestamp of *next rising edge + lookahead.* This optimization can only be applied for the Manifold systems that have the property of generating the events only at the rising edge. Otherwise, events could occur between point *A* and point *B*, and leads to violation of event order and cause problem. So, in general case, when *B* is not safe to process, the timestamp of the null-message can only be *A + lookahead.* Because the positive lookahead value is always be added to the current simulation time to produce the timestamp in null-messages, and such the progresses are guaranteed at all LPs. The deadlock free property and general correctness of SWB algorithms can be easily proved.

5.2.2 The Forecast Null-message Algorithm

The SWB algorithm utilizes the system property of generating the events only at the rising edge and improves the lookahead for a limited amount. However, the SWB algorithm is only one step towards the right direction. From our observation, the components inside a given LP have other sources of delay than the inter-LPs link delay alone. For example, the routers in the interconnection network have a processing delay of 4 clock cycles. In another words, a message can be transfer to other routers or the corresponding cache at least after 4 cycles when the router first see the message. From our perspective, the processing delay of architecture components such as the 4 clock cycles' processing delay of a router is a valuable source of lookahed that could be added on top of the inter-LPs link delay. The processing delay can be utilized by examine the states of components during simulation run time, and make conservative prediction which we called forecast. The forecast is utilized in our Forecast-Null-Message (FNM) algorithm.

The FNM algorithm is fundamentally an improved version of SWB algorithm. It consisted by three major parts, which include:

- A mechanism for components to make conservative prediction based on their states.
- Enhanced null-message that can carry the forecast information.
- A null-message-based synchronization algorithm that can utilize the forecast to improve the lookahead.

For the FNM algorithm if a LP is blocked, every component with inter-LPs link of that LP makes conservative prediction based on the mechanism provided by Manifold for the earliest time it possibly needs to send out a message. After the forecasts from all the components with inter-LPs link are ready, the forecast information is combined, padded into the null-message and sent to relative neighbor LPs. With forecast information from both sides of an inter-LPs link, FNM algorithm is able to make use of the processing delay of the components and improve the lookahead compared to the baseline and SWB algorithm.

The forecast of future messages is not straight forward with the credit based flow control protocol we applied to the system. It can send a credit back whenever it receives a

message from other component. So, the FNM algorithm needs to make the forecast to the future general message as well as the future credit. In the system we studied, there are three points that the forecast should be made, which includes: cache, network interface, and router of interconnection network.

5.2.2.1 Forecast from the Cache

The cache model we use is called MCP-cache [32], which has a two-level coherent cache system runs the MESI protocol. Both L1 and L2 caches has a predefined *lookup time*. The *lookup time* is the delay of a cache to process the incoming request. So, if a cache receives a request at simulation cycle *t*, the earliest possible time it sends out a response is at *t + lookup time*, and the credit is sent back to the sender one cycle after it receives a message.

The system states we examine are mainly the buffer states. And, to generate the forecast for next outgoing message and credit we maintain three additional queues of timestamp besides the buffer state:

- **The outgoing general messages**: for this type of message the forecast is set to *t + lookup time*, where the *t* is the timestamp when the system received the request. We store the timestamps in a priority queue we called *msg_queue.*
- **The credit for processed incoming messages**: for this type of credit, if a general message is processed at *t*, the corresponding credit should be sent out at *t + 1* simulation cycle. The timestamps is saved in another priority queue called *credit_queue.*
- **The credit for unprocessed incoming messages**: the messages could remain unprocessed when the forecast is making. The timestamps for credits of the unprocessed events are not saved in the *credit_queue*. Instead, we modified the simulation kernel to notify the MCP-cache about all the incoming messages, which records down the timestamps of all incoming messages in the *in_msg_queue* and we calculate the forecast base on the timestamps in the *in_msg_queue*.

Based on above description, the logic for MCP-cache to make the forecast can be seen in Algorithm 3. The algorithm is simple and straight forward. It checks the state of the output buffer and three above mentioned priority queues to determine the forecast for the cache.

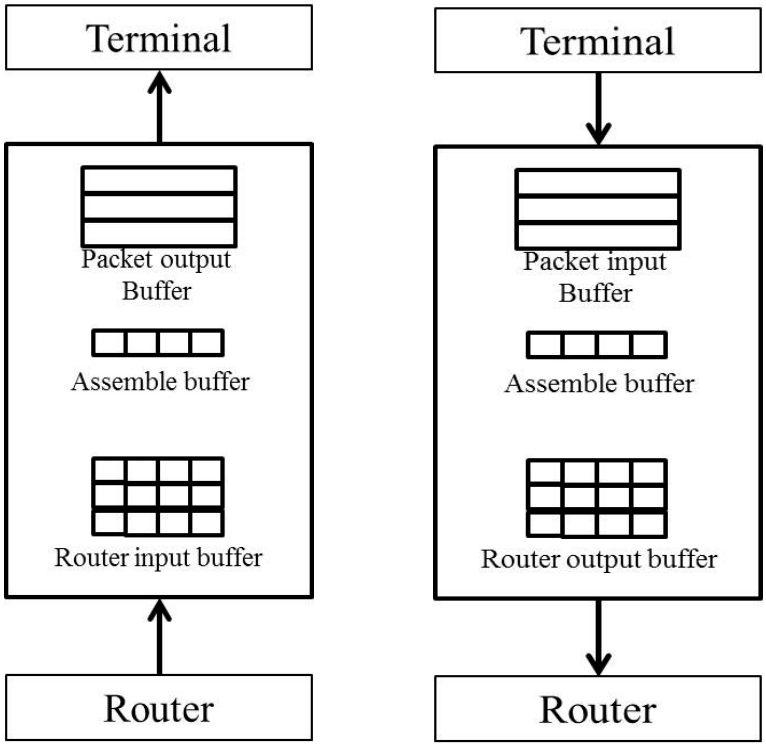
Algorithm 3: outgoing message forecast algorithm of MCP-cache

```
1:  function MCP_FORECAST_NEXT_OUTPUT
2:      now = current cycle
3:      if L1's output buffer not empty || L2's output buffer not empty then
4:          return now
5:      remove obsolete entries from msg_queue and credit_queue
6:      forecast = 0
7:      if there is message or credit scheduled then
8:          forecast = timestamp of earliest message or credit
9:      else
10:         forecast = min(L1 lookup time, L2 lookup time) + now
11:     end if
12:     remove obsolete entries in in_msg_queue
13:     if in_msg_queue not empty then
14:         t = 1 + smallest timestamp of in_msg_queue
15:         forecast = min(forecast, t)
16:     end if
17:     return forecast
18: end function
```

5.2.2.2 Forecast from the Network Interface

The network model we use is Iris, which is a flit-level [33] and cycle-level network model. The Iris network consisted by a certain number of routers and same number of network interface (NI). Each router is connected to one and only one NI. The routers are connected to each other in a way that determined by pre-selected network topology, which in our experiments is Torus topology that can be seen in figure 5. Each NI is connected to one terminal that can be different types such as cache or memory controller. The credit based flow control is applied between the NI and terminal, and among the routers inside the network.

The NI's function includes break the packets that received from the terminal into flits [33], which is the unit that can be transferred over the network. At the same time, it integrates the flits that received from the network into packet when all the flits that belong to a packet are received and sends the packet to terminal. Figure 12 shows the internal structure of the NI, where 12-(a) shows the router-interface-terminal direction's structure and 12-(b) shows the structure of opposite direction. As we can see, the packet in/out buffer holds packets that from/to the terminal. The assemble buffer which is an intermediate buffer that assemble/dissemble the packet into flits. It has one slot per virtual channel (VC) [33]. At the last stage, the router in/out buffer stores the flits from/to each VC. Both the packet and flit transfer is under the credit-based flow control, such that a packet or flit can be delivered to next hop along the path only if there is one or more credits is available for sending.



(a) (b)

Figure 12: Internal structure of Network Interface

The NI has following properties, and we utilized the properties to calculate the forecast of the NI:

- There is one cycles processing delay between the assemble buffer and the packet buffers. Such that if the last flit of packet arrives at the assemble buffer, the earliest time that the packet will be moved to the packet buffer is $t + 1$ cycle.
- The routers send at most one flit to the NI per cycle.
- The router implements a four stages pipeline design. So, it has a processing delay of four cycles. If a flit received by a router, it takes at least four cycles before the router can forward the flit to the NI.
- The NI sends credit back to the terminal after one of the packet from that terminal is moved from the input packet buffer to the assemble buffer.

The process for calculating the forecast of the NI is similar to that of calculating the forecast for MCP-cache, which can be seen in Algorithm 4.

Algorithm 4: outgoing message forecast algorithm of network interface

```
1:  function IRIS_NI_FORECAST_NEXT_OUTPUT
2:      now = current cycle
3:      if outgoing credit scheduled for this cycle then
4:          return now
5:      if output packet buffer not empty then
6:          return now
7:      if an output packet will be assembled this cycle then
8:          return now
9:      when = 0
10:     if there is an input message
11:         m_ts = input message scheduled time
12:         if m_ts < now then
13:             return now
14:         else
15:             when = m_ts - now
16:         end if
17:     else
18:         for each virtual channel v do
19:             if asm_buf[v] has flits then
20:                 forecast[v] = packet length – received flits
21:             else
22:                 if router_in_buf[v] has flits then
23:                     forecast[v] = packet length
24:                 else
25:                     forecast[v] = router→earliest() + 2
26:                 end if
27:             end if
28:         end for
```

```
29:                 when = min(forecast[])
30:         end if
31:         return now + when
32: end function

33: function ROUTER::EARLIST
34:         if there is one in-transit packet bound for NI then
35:                 return 0
36:         else
37:                 return 4
38:         end if
39: end function
```

The lines 1-8 of Algorithm 4 are self-explanatory. The variable *when* at line 8 is the possible time for future message, and it is relative to now. Lines 9-15 handle the credit for incoming messages from the terminal. We implemented a FIFO queue to assist the handling of credit. Whenever an incoming message arrives, its schedule time is pushed back to the end of queue. And, when a credit is sent the first item of the queue is removed. If there is a un-credit incoming message scheduled at *m_ts*, if *m_ts* < *now*, which indicates the message is scheduled in past, then the credit could be sent at any time from *now* (line 13). Otherwise, the credit could be sent at any time after *m_ts – now* (line 14).

The Lines 18-29 deal with the case that there is no credit to be sent. In this case, we only forecast the next out-going non-credit message that receives from the router side. We first check whether there is anything in the assemble buffer (line 19). If there is, since the header flit that contains the length of the packet must be the first flit arrives, we can acquire the length of packet from header flit regardless the states of remaining flits. As it can transfer at most one flit per cycle, we can safely forecast the next outgoing non-credit message from the particular VC can be sent no sooner than *now* + *packet length – number of received flits* (line 20). On the other hand, if there is nothing in the assemble buffers. We check the input flit buffers (line 22). If the input flit buffer for a VC is not

empty, it must hold the header flit as the assemble buffer is empty. In this case we predict the next outgoing message should be sent no earlier than *packet length* + *now* (line 23). Otherwise, the NI sends request to the router to consult the earliest possible time for next incoming flit for the given VC. As it takes at least one cycle to transfer the next flit to input flit buffer, and one cycle to move the flit to assemble buffer, we added two cycles to the forecast (line 25). At line 29, we determine the forecast by selecting the minimum forecast from all the VC.

Lines 33-39 is the corresponding router's function that returns the *earliest*, which be invoked at line 25. It returns 0, if there is a flit for a given VC of NI. And, it returns 4, if there is not because the 4 cycles' processing delay of the router.

5.2.2.3 Forecast from the Router

As mentioned above, the router of interconnection network has a 4 stages pipeline, which includes:

- Route computing (RC)
- Virtual channel allocation (VA)
- Switch allocation (SA)
- Switch traversal (ST)

To transfer a packet, if the entire packet can be sent over a single flit, then only one flit is required. Otherwise, at least three flits are needed: a header flit, one or more body flit, and a tail flit. Each header flit must go through all four stages of the router's pipeline, while other two types of flit only go through two stages: the SA and ST.

A VC can in one of the following five states. The states indicate the state of the packet that occupying that VC.

- **EMPTY**: the VC is not occupied
- **FULL**: the header flit of a packet is entered the VC
- **VCA_REQUESTED**: the route computation is completed, and the header flit is waiting the output VC allocation
- **SWA_REQUESTED**: flit is waiting for use the switch.
- **SW_TRAVERSAL**: the flit is using the switch

The Algorithm 5 shows the forecast procedure at the router, it utilizes the states information of the router to calculate the forecast. The procedure is invoked by each router when the forecast update is needed. We defined the border port as the port that connected to an inter-LPs link. For each router, we calculate the forecast for every border port separately.

Algorithm 5: outgoing message forecast algorithm of Iris router

```
1:  procedure IRIS_ROUTER_FORECAST_NEXT_OUTPUT
2:      for each port p that connected to an inter-LPs link do
3:      for each port q do
4:          if p == q then
5:              for each VC v of port p do
6:                  switch state of v
7:                      case EMPTY or FULL
8:                          vc_pred = PIPE_DEPTH - 1
9:                      case VCA_REQUESTED
10:                         vc_pred = 2
11:                     case SWA_REQUESTED
12:                         vc_pred = 1
13:                     case SW_TRAVERSAL
14:                         vc_pred = 0
15:                 end switch
16:                 if vc_pred < pred then
17:                     pred = vc_pred
18:                 end if
19:             end for
20:         else
21:             for each VC v of part q do
```

```
22:                     if state of v is not EMPTY or FULL || output port of v is
                        not p then
23:                             vc_pred = PIPE_DEPTH
24:                     else
25:                         switch state of v do
26:                             case EMPTY or FULL
27:                                 vc_pred = PIPE_DEPTH - 1
28:                             case VCA_REQUESTED
29:                                 vc_pred = 2
30:                             case SWA_REQUESTED
31:                                 vc_pred = 1
32:                             case SW_TRAVERSAL
33:                                 vc_pred = 0
34:                             end switch
35:                         end if
36:                         if vc_pred < pred then
37:                             pred = vc_pred
38:                     end for
39:                 end for
40:                 forecast value = pred + now
41:             end for
42:     end procedure
```

As we can see from Algorithm 5, two cases are considered separately. In the first case, we deal with the credits for incoming flits at the border port, and in the second case we check the outgoing messages at other port that can go through the border port. Lines 5-19 handle the first case. We check the state of each VC to determine the earliest timestamp that a flit will leave the router and a credit will be sent. The *PIPE_DEPTH* is the number of stages, if the VC is in EMPTY or FULL state the earliest time that a credit could be sent is *PIPE_DEPTH* – 1 (line 8). If the VC state is in VCA_REQUESTED, SWA_REQUESTED or SW_REQUESTED, the soonest outgoing credit could be sent

within next 2, 1 or current cycle, respectively. Lines 21-38 handle the second case. If the VC is occupied by packet that will not go through the border port *p*, then it is safe to guarantee a forecast for *PIPE_DEPTH* (line 23) as it needs to finish transfer of current packet first. Otherwise, we calculate the forecast follows the same rules as that for first case

5.2.2.4 The Synchronization Algorithm

We presented the forecast algorithm for components in above sections. In this section we illustrate our Forecast null-message synchronization algorithm. The FNM algorithm utilize the forecast generates by each component and improve the lookahead. To carry the forecast of components, we slightly enhanced the null-message that allows it to carry the forecast along with the timestamp that generated by general algorithm. The enhanced null-message can be seen in 13, the *ts_1/ts_2* is the timestamp generated by general algorithm, and the *forecast_1/forecast_2* is the forecast from LP1 and LP2, respectively. The forecast is essential a guaranteed time that a LP will not generate any cross LP event before. With knowing the forecast from both LPs that share an inter-LPs link, both LPs can safely make the prediction that within the timestamp (*now*, *minimum forecast*) there will not be any activity on the inter-LPs link. So, the events have timestamp within that periodic can be safely processed.

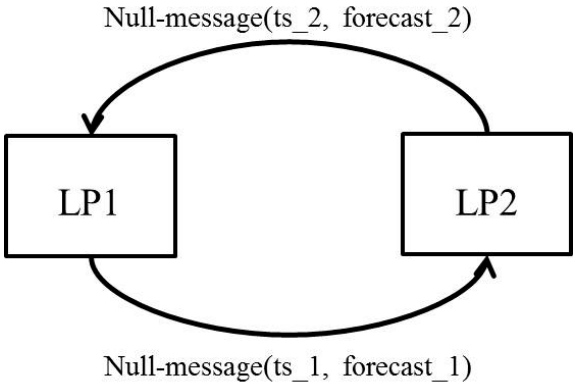


Figure 13: Enhanced null-message with forecast

The Algorithm 6 presents the calculation of forecast null-message algorithm. The lines 2-9 are the same procedure as the SWB algorithm. Lines 10-12 are where the forecast is made for each component that has an inter-LPs link. The function *forecast_next_output()* is invoked at each component when the execution is blocked, and this function returns the forecast of that component. In the next FOR loop (line 14-22), it calculate the timestamp for next null-message. Inside the FOR loop, the algorithm compare the forecast from a given neighbor (*in_forecast*) with the local forecast (*out_forecast*) and find the minimum between *out_forecast* and *in_forecast + link_delay* (line 17). The minimum is compared to the *nextClockTime* and the maximum between these two values is selected as the candidate for timestamp in next null-message (line 18), from this line we can see that the lower bound of lookahead of FNM algorithm is the lookahead of SWB algorithm. The minimum value from the candidate is picked as the *min_null_ts* that will be sent out in the null-message (line 19-21). In lines 23-27 the null-message that contains the timestamp as well as the forecast is composed and sent to corresponding neighbor.

Algorithm 6: the Forecast Null-message algorithm

```
1:  while simulation not terminated do
2:      receive messages
3:      nextClock = clock whose next edge has smallest timestamp among all clocks
4:      nextClockTime = simulation time of next edge of nextClock
5:      receive null-messages
6:      min_null = minimum timestamp of the null-messages
7:      if nextClockTime < min_null then
8:          process all events scheduled for the clock edge
9:      else
10:         for each border component c do
11:             c → forecast_next_output()
12:         end for
```

```
13:                 min_null_ts = 0
14:                 for each neighbor n do
15:                 out_forecast[n] = timestamp of next possible outgoing message to
                    the neighbor
16:                 in_forecast[n] = timestamp of next possible incoming message
                    from the neighbor
17:                         forecast = min(out_forecast, in_forecast + link_delay)
18:                         null_ts = max(forecast, nextClockTime) + lookahead
19:                         if min_null_ts == 0 || min_null_ts > null_ts then
20:                                 min_null_ts = null_ts
21:                         end if
22:                 end for
23:                 for each neighbor n do
24:                         if min_null_ts > last min_null_ts then
25:                                 send null-message(min_null_ts, out_forecast[n])
26:                         end if
27:                 end for
28:         end if
29: end while
```

Since lookahead and forecast are always positive, the conservative messages for a given inter-LPs link are sent and received in the correct order, and the events are processed only when it is guaranteed to be safe, it is not hard to prove the general correctness and deadlock free property of the FNM algorithm.

5.3 Design of Experiments

We implemented all the baseline algorithm, SWB and FNM algorithms. Due to the limitation for the hardware resources we discuss in above section, we performed comprehensive tests for the only the SWB and FNM algorithms, while just performed

several preliminary tests for baseline algorithm. The tests were conducted on the same Linux cluster as that we used in tests for the partition schemes and the proxy design.

For the SWB and FNM algorithms, as the Manifold supports both the trace-driven and timing-directed simulation, we performed tests for both types of simulation. The system models we tested include 16-, 32-, 64- and 128-core systems. In the tests, we applied the same cache and interconnection network models, and the credit-based flow control mechanism as what we used in last two chapters. The cache's processing delay is set to 5 clock cycle. We test our system models against five randomly chosen PARSEC 2.0 benchmarks. In all of our tests, the y-part scheme for the network is applied. Tests for each system model, simulation type and benchmark combination are repeated for 3 times to deal with the randomness introduced by the un-controllable network traffic of the Linux cluster. The experimental results we present in the following sections are the average value over 3 simulation runs.

In the tests for *trace-driven* simulation, we followed the same LP assignment as what we did in Chapter 2. While in the tests for *timing-directed* simulation, we employed the proxy design and followed the same LP assignment as that mentioned in Chapter 3. In our experiments, *trace-driven* and *timing-directed* simulations do not produce the exact same experimental results. The non-determinism is resulting from the ordering of events in the same clock cycle. However, the variance is very small, and the variance does not affect the correctness of the simulation.

5.4 Evaluation

Our test results show clearly the effects that synchronization algorithms can have on the performance of parallel simulation of multi-core systems. The FNM algorithm significantly reduced the number of null-messages, and for both *trace-driven* and *timing-directed* simulations, it consistently achieves the best performance and scalability compared to the baseline and SWB algorithm.

5.4.1 Evaluation for Trace-driven Simulation

Tables 16 through 19 present the test results for number of null-messages generated per simulation cycle of *trace-driven* simulation. Each algorithm has one column. The numbers outside the parentheses show the number of null-messages generated system wide for one simulation cycle, and the numbers inside the parentheses present the number of null-messages that generated per inter-LPs link during one simulation cycle. The third column shows the improvement of FNM algorithm against the SWB algorithm.

In Table 16, we can see the FNM algorithm generates about 0.3 null-messages per simulation cycle for each inter-LPs link in tests for 16-core system model. Compared to the SWB algorithm it has from 38% to 49.8% improvement. When the system scale reached 32-core and above, SWB algorithm generates 0.5 null-messages per-inter-LPs link per-simulation-cycle for all benchmarks with very little difference, which can be seen in Tables 17 through 19. On the other hand, in tests for 32-, 64- and 128-core system models, for four out of five benchmarks, the FNM algorithm constantly generates 0.3 null-messages per-inter-LPs link per-simulation-cycle, and achieves over 40% improvement compared to the SWB algorithm. In tests for *ferret* benchmark, the FNM generates roughly 0.4 null-messages per-inter-LPs link per-simulation-cycle, which is more than what it has for the other four benchmarks. However, it still has from 20.2% to 24.4% less null-messages against the SWB algorithm. To summarize our observation, the FNM algorithm significantly reduced the number of null-messages generated during the simulation compared to the SWB algorithm.

Table 16: Number of Null-messages Per Cycle for 16-core model (trace-driven)

Benchmark	SWB	FNM	improvement
facesim	23.78 (0.57)	13.42 (0.32)	43.60%
ferret	24.02 (0.57)	14.9 (0.35)	38.00%
freqmine	23.99 (0.57)	13.11 (0.31)	45.40%
streamcluster	24.45 (0.58)	12.67 (0.3)	48.20%
vips	24.48 (0.58)	12.29 (0.29)	49.80%

Table 17: Number of Null-messages Per Cycle for 32-core model (trace-driven)

Benchmark	SWB	FNM	improvement
Facesim	38 (0.5)	22.82 (0.3)	39.90%

Ferret	38 (0.5)	30.31 (0.4)	20.20%
Freqmine	38 (0.5)	22.21 (0.29)	41.60%
streamcluster	38 (0.5)	21.21 (0.28)	44.20%
Vips	38 (0.5)	20.62 (0.27)	45.70%

Table 18: Number of Null-messages Per Cycle for 64-core model (trace-driven)

Benchmark	SWB	FNM	improvement
Facesim	72 (0.5)	43.46 (0.3)	39.60%
Ferret	72 (0.5)	54.55 (0.38)	24.20%
Freqmine	72 (0.5)	42.05 (0.29)	41.60%
streamcluster	72 (0.5)	40.71 (0.28)	43.50%
Vips	72 (0.5)	40.35 (0.28)	44.00%

Table 19: Number of Null-messages Per Cycle for 128-core model (trace-driven)

Benchmark	SWB	FNM	improvement
Facesim	140 (0.5)	95.53 (0.34)	31.80%
ferret	140 (0.5)	105.9 (0.38)	24.40%
freqmine	140 (0.5)	94.21 (0.34)	32.70%
streamcluster	140 (0.5)	86.2 (0.31)	38.40%
vips	140 (0.5)	90.8 (0.32)	35.10%

The performance in terms of total time consumption to run the simulation can be seen in the Tables 20-23. The first column of the tables shows the benchmark name, the second column shows the time consumption of sequential simulation in seconds. Each algorithm has its own column, inside the column the numbers outside the parentheses are the total time consumption in seconds, while the numbers inside the parentheses are the speedup against the sequential simulation. The last column of the tables presents the improvement of FNM algorithm compared to the SWB algorithm.

Table 20 shows the test results for 16-core system model, as we can see the SWB algorithm has from 5.5 to 6.2 times speedup compared to the sequential simulation. The FNM algorithm achieves from 6.3 to 7.0 times speedup against the sequential simulation, and has from 10.2% to 11.9% improvement over the SWB algorithm. Both algorithms achieve better speedup against the sequential simulation in tests for 32-core system model compared to what they had in tests for 16-core system model. As the data in Table 21 shows, the speedup of SWB algorithm is from 5.8 to 6.3 in tests for 32-core system model, which improved from 0.2% to 4% against the tests of 16 core system model. At

the same time the speedup of FNM algorithm is from 7.3 to 8.6, which are from 15% to 24% better compared to the tests for 16-core system model. In comparison between the two algorithms, the FNM algorithm has from 21% to 27% improvement over the SWB algorithm.

In tests for 64-core and 128-core system models, the speedup that both algorithms have against the sequential simulation kept increasing. In tests for 64-core systems, SWB algorithm has about 9 times speedup and FNM algorithm has over 10.8 times speedup against the sequential simulation. In tests for 128-core systems, the speedup against the sequential simulation grows to 12.5-13.5 and 15.9-17.1 times for the SWB and FNM algorithm, respectively. If we compare the two algorithms, we can see in tests for 64-core and 128-core system models, the FNM algorithm always outperforms the SWB algorithm. It outperforms the SWB algorithm by 16%-29% in tests for 64-core system model, and 21%-25% in tests for 128-core system model.

Table 20: Simulation Run Time in Seconds for 16-core model (trace-driven)

Benchmark	Seq.	SWB	FNM	improvement
Facesim	4037	658 (6.1×)	582 (6.9×)	11.60%
Ferret	4017	727 (5.5×)	642 (6.3×)	11.70%
Freqmine	4079	662 (6.2×)	583 (7.0×)	11.90%
streamcluster	4124	678 (6.1×)	607 (6.8×)	10.50%
Vips	4081	654 (6.2×)	587 (7.0×)	10.20%

Table 21: Simulation Run Time in Seconds for 32-core model (trace-driven)

Benchmark	Seq.	SWB	FNM	improvement
Facesim	8813	1379 (6.4×)	1021 (8.6×)	26.00%
Ferret	8186	1418 (5.8×)	1117 (7.3×)	21.20%
Freqmine	8863	1408 (6.3×)	1032 (8.6×)	26.00%
streamcluster	8489	1392 (6.1×)	1033 (8.2×)	25.80%
Vips	8723	1383 (6.3×)	1008 (8.7×)	27.10%

Table 22: Simulation Run Time in Seconds for 64-core model (trace-driven)

Benchmark	Seq.	SWB	FNM	improvement
Facesim	18995	2155 (8.8×)	1521 (12.5×)	29.40%
Ferret	18426	2025 (9.1×)	1700 (10.8×)	16.00%
Freqmine	19393	2128 (9.1×)	1510 (12.8×)	29.00%

streamcluster	19094	2152 (8.9×)	1556 (12.3×)	27.70%
Vips	18942	2090 (9.1×)	1601 (11.8×)	23.40%

Table 23: Simulation Run Time in Seconds for 128-core model (trace-driven)

Benchmark	Seq.	SWB	FNM	improvement
Facesim	29546	2362 (12.5×)	1809 (16.3×)	23.40%
ferret	29294	2342 (12.5×)	1843 (15.9×)	21.30%
freqmine	30086	2351 (12.8×)	1826 (16.5×)	22.30%
streamcluster	29247	2301 (12.7×)	1714 (17.1×)	25.50%
vips	31491	2339 (13.5×)	1837 (17.1×)	21.50%

5.4.2 Evaluation for Timing-directed Simulation

In tests for timing-directed simulation, we also recorded down the number of null-messages generated during simulation and the total simulation run time. In our tests, the simulation was run for 200 million cycles. The test results are presented in the following tables in a same manner as that in last section.

Tables 24-27 show the total number of null-messages generated per-simulation-cycle and average number of null-messages generated per-inter-LPs link per-simulation-cycle. As we can see from the Table 24, the FNM algorithm reduced the number of null-messages by more than 30% against the SWB algorithm in tests for 16-core system. Table 25, 26 and 27 present the test results for 32-, 64-, and 128-core system, respectively. From the test results we can see SWB algorithm generated at least 21.5%, 44.7% and 11.3% more null-messages than the FNM algorithm in tests for 32-, 64-, and 128-core system model, respectively. So, for both *trace-driven* and *timing-directed* simulation, the FNM algorithm significantly reduced the number of null-messages generated during the simulation compared to SWB algorithm.

Table 24: Number of Null-messages Per Cycle for 16-core model (timing-directed)

Benchmark	SWB	FNM	improvement
Facesim	15.07 (0.58)	9.98 (0.38)	33.80%
Ferret	15.11 (0.58)	10.01 (0.39)	33.80%
Freqmine	15.08 (0.58)	9.31 (0.36)	38.30%
Streamcluster	15.15 (0.58)	9.12 (0.35)	39.80%
Vips	15.07 (0.58)	9.41 (0.36)	37.60%

Table 25: Number of Null-messages Per Cycle for 32-core model (timing-directed)

Benchmark	SWB	FNM	improvement
Facesim	22 (0.5)	16.83 (0.38)	23.50%
Ferret	22 (0.5)	17.28 (0.39)	21.50%
Freqmine	22 (0.5)	16.57 (0.38)	24.50%
Streamcluster	22 (0.5)	16.15 (0.37)	26.60%
Vips	22 (0.5)	16.57 (0.38)	24.70%

Table 26: Number of Null-messages Per Cycle for 64-core model (timing-directed)

Benchmark	SWB	FNM	improvement
Facesim	75.44 (0.88)	40.74 (0.46)	46.00%
Ferret	74.13 (0.84)	40.98 (0.47)	44.70%
Freqmine	74.56 (0.85)	41.04 (0.47)	45.00%
Streamcluster	72.78 (0.83)	39.67 (0.45)	45.50%
Vips	73.93 (0.84)	40.57 (0.46)	45.10%

Table 27: Number of Null-messages Per Cycle for 128-core model (timing-directed)

Benchmark	SWB	FNM	improvement
facesim	76 (0.5)	67.37 (0.44)	11.40%
ferret	76 (0.5)	67.43 (0.44)	11.30%
freqmine	76 (0.5)	67.44 (0.44)	11.30%
streamcluster	76 (0.5)	67.26 (0.44)	11.50%
vips	76 (0.5)	67.22 (0.44)	11.60%

Tables 28-31 present the time consumption to run the simulation. The tables are also in the same format as last section. The only difference is that the time consumptions are presented in minutes instead of in seconds. From Table 28 we could see the FNM algorithm reduced time consumption by over 10% against the SWB algorithm, and it achieves from 4.9 times to 5.5 times speedup compared to the sequential simulation. The table 29 presents the test results for 32-core systems. In tests for 32-core system model, compared to the test results for 16-core system model, the performance gap between SWB and FNM algorithms increased from 10.2%-14.7% to 23.5% to 27.5%, and the speedup of FNM algorithms has against the sequential simulation also grows to 6.7-8.6 times.

As shown in Table 30 and 31, in tests for 64-core system model, the FNM algorithm consumes from 24.2% to 26.5% less time to run the simulation than the SWB

algorithm. Compared to the sequential simulation SWB algorithm achieves 5.6-9.1 times speedup and FNM algorithm achieves 7.6-12.1 times speedup. Finally, in the tests for 128-core system model, the speedup of FNM algorithm has against the sequential simulation is 15.5-19.1 times, which is from 12% to 15.4% better than that of SWB algorithm. For both algorithms the speedup against sequential simulation grows with system scale, and they show reasonable scalability on cluster-based parallel hardware for parallel simulation of multi-core systems.

Table 28: Simulation Run Time in Minutes for 16-core model (timing-directed)

Benchmark	Seq.	SWB	FNM	improvement
facesim	1259.3	261.6 (4.8×)	234.9 (5.4×)	10.20%
ferret	1124.8	254.7 (4.4×)	227.8 (4.9×)	10.60%
freqmine	1203.3	255.5 (4.7×)	218 (5.5×)	14.70%
streamcluster	1183.8	252.8 (4.7×)	222.7 (5.3×)	11.90%
vips	1167	257.5 (4.5×)	227.3 (5.1×)	11.70%

Table 29: Simulation Run Time in Minutes for 32-core model (timing-directed)

Benchmark	Seq.	SWB	FNM	improvement
facesim	2614.2	397.3 (6.6×)	303.6 (8.6×)	23.60%
ferret	1777.9	343.3 (5.2×)	255.6 (7.0×)	25.50%
freqmine	1635.6	331.5 (4.9×)	245.6 (6.7×)	25.90%
streamcluster	1710.6	336.8 (5.1×)	244.3 (7.0×)	27.50%
vips	1716.3	336.3 (5.1×)	257.2 (6.7×)	23.50%

Table 30: Simulation Run Time in Minutes for 64-core model (timing-directed)

Benchmark	Seq.	SWB	FNM	improvement
facesim	3170.2	465.8 (6.8×)	342.3 (9.3×)	26.50%
ferret	2534.3	449.4 (5.6×)	331.3 (7.6×)	26.30%
freqmine	2718.9	444.9 (6.1×)	337.3 (8.1×)	24.20%
streamcluster	4796.4	526.6 (9.1×)	396.2 (12.1×)	24.80%
vips	2564.6	446.7 (5.7×)	337.9 (7.6×)	24.40%

Table 31: Simulation Run Time in Minutes for 128-core model (timing-directed)

Benchmark	Seq.	SWB	FNM	improvement
facesim	7097.5	540.5 (13.1×)	459.1 (15.5×)	15.10%
ferret	8212.7	542.9 (15.1×)	474.6 (17.3×)	12.60%
freqmine	8359.1	539.6 (15.5×)	474 (17.6×)	12.20%

streamcluster	9241.7	548.6 (16.8×)	482.9 (19.1×)	12.00%
vips	7471.7	537.2 (13.9×)	454.3 (16.4×)	15.40%

5.5 Discussion

The application-dependent optimization for null-message-based synchronization algorithm demonstrated significant improvement over the traditional null-message algorithms. The Forecast Null-message algorithm that utilizes the domain-specific knowledge that acquire from the architecture components consistently achieves better performance than the Send-When-Block algorithm that merely utilizes the system attribute that messages are sent only at the rising edge. For both SWB and FNM algorithm, the improvement of performance can be traced to the improved lookahead and reduced number of null-messages. By using the domain-specific knowledge to calculate the lookahead dynamically during the simulation FNM algorithm has significantly greater lookahead than the traditional algorithm and also reduced the number of null-messages. The SWB algorithm that utilizing only one system attribute also slightly improved the lookahead and reduced the number of null-messages. Our tests show that both application-dependent optimized algorithms achieve reasonable scalability on cluster-based parallel hardware, and we believe the application-dependent optimizations have potential to be applied in other domains of parallel simulation.

CHAPTER 6

CONCLUSION AND FUTURE WORKS

This dissertation proposed and evaluated technologies to optimize the parallel simulation of multi-core systems. In this conclusive chapter the contributions of above chapters are summarized and the possible future works are discussed.

6.1 Contributions

In chapter 3 we studied the effects of partitioning schemes can have on the null-message-based parallel simulation of multi-core systems. We design and examined three manual partitioning schemes in our study, which including the 1-part, 2-part and y-part, they differ from each other only on how the interconnection network of the multi-core system is partitioned. From our test results, we observed that the partitioning schemes can have significant effects on the performance and parallel efficiency. The y-part that divided the interconnection network into Y equal pieces, where Y is the *y* dimension of an X×Y Torus network, consistently achieves the best performance in terms of time consumption. The y-part also achieves better parallel efficiency when the system scale reached 32-core and above. From our experimental result, we found that the time network LP(s) spends on gathering and processing the null-messages is the major reason that leads to the difference in performance and parallel efficiency. By partitioning the network into more pieces, it reduced the null-message gathering and processing time and leads to better performance. Based on this observation, we concluded that in a null-message-based parallel simulation, any component that has growing number of inter-LPs links with increasing system scale should, in theory, be partitioned to achieve reasonable performance and parallel efficiency.

In Chapter 4, we demonstrated a technology to improve the performance of *timing-directed* parallel simulation of multi-core system. The technology we proposed is designed to hide the TCP/IP communication delay from the back-end's perspective. It essentially adds an intermediate level called Proxy between the functional *front-end* and timing *back-end*. The Proxy acquires instructions from the front-end and stores the

instructions in shared memory segments to allow the efficient access from the back-end. From our experimental results, we found that the Proxy design greatly improved the overall performance in terms of simulation run time compared to the original design. The reason for the difference in performance can be traced to the chain effect that incited by the instruction retrieving latency.

Chapter 5 presented a new application-dependent optimized null-messages algorithm called Forecast Null-message algorithm. Differ from the traditional null-message optimizations that completely independent from the simulation components, the FNM algorithm utilizes the domain-specific knowledge that acquired from the architecture components and significantly improved the lookahead. In comparison with traditional algorithm and the Send-When-Block algorithm, it greatly reduced the number of null-messages generated during simulation and achieves better performance in both *trace-driven* and *timing-directed* simulation. Additionally, with system scale grows FNM algorithm demonstrated reasonable scalability on cluster-based parallel hardware with growing speedup against the sequential simulation.

With the optimizations mentioned above, for the system model we are interested in, the parallel simulation of Manifold achieved competitive speedup against sequential simulation compared to other parallel simulator. Reinhardt and colleagues didn't provide the speedup of WWT against sequential simulation. The SST achieves roughly 6x-12x and 8x-16x speedup against sequential simulation in tests use 16 and 32 shared memory ranks [47], respectively. The parallel simulation of Manifold use similar number of ranks has speedup roughly equals to the worse case of SST. However, the system model tested in [47] has at least 8 thousands of simulation nodes, which fundamentally different from the system model we tested. Additionally, the tests for SST run with shared memory ranks which are also different from the MPI-based ranks we use for parallel simulation of Manifold. The SlackSim achieves about 2.1 times speedup against sequential simulation in cycle-by-cycle simulation for 8 host cores system model [46], which is lower than the more than 3x speedup we have for similar simulation with Manifold platform.

6.2 Future Works

The partitioning schemes we studied are only manual partitioning schemes. The future generation of multi-core and many-core system can have huge number of components and very high complexity. To simulation such system a complex simulation program is theoretically necessary, but manual partitioning schemes might not enough to efficiently partition such programs. The automatic partitioning tools are widely used by both researchers and industrial engineers in simulation of large scale system, and it is approved to have the capability of greatly improve the performance of relative parallel simulation. So, one possible future work that can be conducted is applying the automatic partition tool to the null-message based parallel simulation of multi-core system.

The current implementation of QSim server is multi-thread-based, however when the system scales reached a certain level the computing power of a single node might not be enough to server all the cores of the back-end, and could potentially be the bottleneck of the overall performance. Based on this observation, one possible future work could be that implement a distributed version of the QSim server that can utilize the LP-level parallelism.

The current design of FNM algorithm utilizes only the processing delay of cache, network interface and a single router. It can be further improved with additional domain-specific knowledge. For example, the core model we used is a deep pipelined model. It has over 20 pipeline stages which indicate that the minimum processing delay for each instruction is more than 20 cycles. Based on the observation that most instructions do not generate inter-LPs events, with the domain-specific knowledge from the core model, we can further improve the lookahead and performance.

The current implementation of Manifold project can utilize only the LP-level parallelism [38]. For a program that uses the LP-level parallelism, the program is divided into LPs, and LPs need to send and receive the data via inter-processes messages, which typically requires the TCP/IP or similar network protocol to ensure the successful message passing. Due to the overhead of network protocols, the latency of the inter-processes message is in scale of microsecond [33]. On the other hand, the data sharing for programs utilize the thread-level parallelism [37] is simple. Different threads can access the same segment of shared memory with latency from about ten to hundreds of

nanoseconds or so [34]. But the shared memory segments are not accessible by remote nodes on a cluster, though cutting edge work-station or blade of cluster nowadays has at most 72 hardware threads in total [35]. The computational power of 72 hardware threads seem not enough to handle the complex parallel programs those can have components in the order of hundreds of thousands or even more. To better utilize the cluster based parallel hardware, a possible future work could be that employ the hybrid parallelism [44] design for Manifold project. A high level view of the system with hybrid parallelism can be seen in the Figure 14, the simulator with hybrid parallelism uses the thread-level parallelism at each node, while allow inter-nodes message exchange with MPI and utilize the LP-level parallelism.

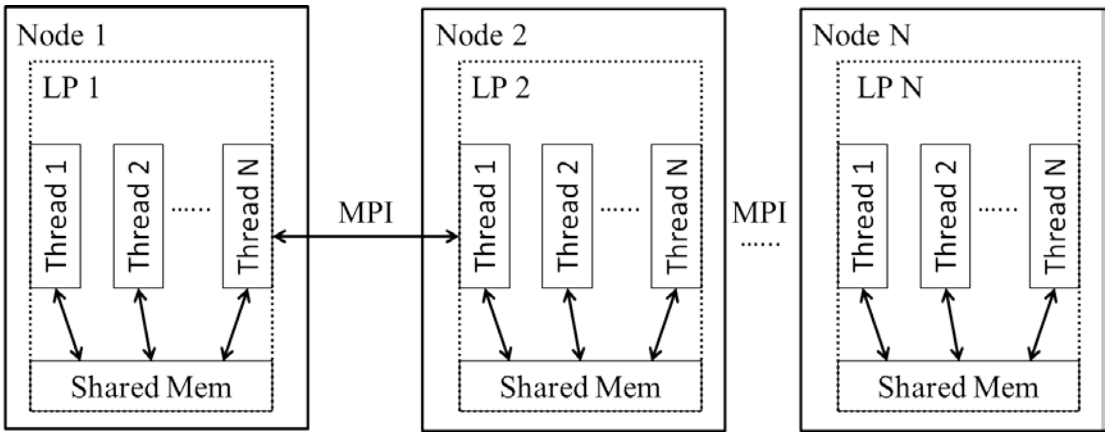


Figure 14: Simulation with hybrid parallelism

At last, the idea of utilizing the domain-specific knowledge can be generalized to other domains of parallel simulation. One possible future work is implementing the FNM algorithm for the network simulators such as Network Simulator 3 [45].

REFERENCES

[1] S.W. Keckler, K. Olukotun, and H.P. Hofstee (Eds.). 2009. Multicore Processors and Systems. Springer

[2] Z. Dong, J. Wang, G. Riley, and S. Yalamanchili. 2013. A Study of the Effect of Partitioning on Parallel Simulation of Multicore Systems. In IEEE 21st International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS). 375–379.

[3] J. Wang, J. Beu, R. Bheda, T. Conte, Z. Dong, C. Kersey, M. Rasquinha, G. Riley, W. Song, H. Xiao, P. Xu, and S. Yalamanchili. 2014. Manifold: A Parallel Simulation Framework for Multicore Systems. In 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). 106–115.

[4] A.F. Rodrigues, K.S. Hemmert, B.W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B.Jacob. 2011. The structural simulation toolkit. ACM SIGMETRICS Performance Evaluation Review 38, 4 (March 2011), 37–42.

[5] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," Proceedings of the 16th International Symposium on High-Performance Computer Architecture, pp. 1–12, 2010.

[6] J. Chen, L.K. Dabbiru, D. Wong, M. Annavaram, and M. Dubois. 2010. Adaptive and speculative Slack Simulations of CMPs on CMPs. In Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. 523–534.

[7] R.M. Fujimoto. 2000. Parallel and Distributed Simulation Systems. John Wiley & Sons. January 3, 2000

[8] J. Fan, J. Zhan and X. Zhao "Performance analysis and optimization of MPI collective operations on multi-core clusters", The Journal of Supercomputing, April 2012, Volume 60, Issue 1, pp 141-162

[9] Fujimoto, R. M., Tsai, J.-J., and Gopalakrishnan, G. C., "Design and evaluation of the rollback chip: Special purpose hardware for time warp," IEEE Trans. Comput., vol. 41, pp. 68–82, Jan. 1992.

[10] Mattern, F., “Efficient algorithms for distributed snapshots and global virtual time approximation,” J. Parallel Distrib. Comput., vol. 18, pp. 423–434, Aug.1993.2.

[11] Vulov, G., Hou, C., Vuduc, R., Fujimoto, R., Quinlan, D., and Jef- ferson, D., The backstroke framework for source level reverse computation applied to parallel discrete event simulation,” in Simulation Conference (WSC), Proceedings of the 2011 Winter, pp. 2960–2974, Dec 2011.

[12] Brooks, III, E. D., “The butterfly barrier,” Int. J. Parallel Program., vol. 15, pp. 295–307, Oct. 1986.

[13] Lin, Y.-B. and Lazowska, E. D., “Determining the global virtual time in a distributed simulation.,” in ICPP (3) (Yew, P.-C., ed.), pp. 201–209, Pennsylvania State University Press, 1990.

[14] Mattern, F., “Efficient algorithms for distributed snapshots and global virtual time approximation,” J. Parallel Distrib. Comput., vol. 18, pp. 423–434, Aug.1993.

[15] Rizvi, S., Elleithy, K., and Riasat, A., “Trees and butterflies barriers in distributed simulation system: A better approach to improve latency and the processor idle time,” in Information and Emerging Technologies, 2007. ICIET 2007. International Conference on, pp. 1–6, July 2007.

[16] Bryant, R. E., “Simulation of packet communication architecture computer systems,” tech. rep., Cambridge, MA, USA, 1977.

[17] Chandy, K. and Misra, J., “Distributed simulation: A case study in design and verification of distributed programs,” Software Engineering, IEEE Transac- tions on, vol. SE-5, pp. 440–452, Sept 1979.

[18] T. F. Leibfried Jr., “Deadlock Detection and Recovery Algorithm Using the Formalism of a Directed Graph Matrix,” ACM SIGOPS Operating Systems Review, Volume 23 Issue 2, April 1989

[19] Rassul Ayani and Hassan Rajaei, “Parallel Simulation Using Conservative Time Windows,” Proceeding WSC’ 92 Proceeding of the 24th conference on winter simulation, Pages 709-717

[20] Misra, J., “Distributed discrete-event simulation,” ACM Comput. Surv., vol. 18, pp. 39–65, Mar. 1986.

[21] http://glaros.dtc.umn.edu, “hmetis - hypergraph and circuit partitioning,” http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview.

[22] S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood. 1993. The Wisconsin Wind Tunnel: virtual prototyping of parallel computers. In Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. 48–60.

[23] M. Chidester and A. George. 2002. Parallel simulation of chip-multiprocessor architectures. ACM Transactions on Modeling and Computer Simulation 12, 3 (July 2002), 176–200.

[24] John L. Hennessy; David A. Patterson (16 September 2011). Computer Architecture: A Quantitative Approach. Elsevier. pp. B–12. ISBN 978-0-12-383872-8. Retrieved25 March 2012.

[25] M. S. Papamarcos and J. H. Patel, “A low-overhead coherence solution for multiprocessors with private cache memories,” ISCA’98 25 years of the international symposia on Computer architecture, pp. 284–290, 1998.

[26] C. Bienia and K. Li, “Parsec 2.0: A new benchmark suite for chipmultiprocessors,” Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation, 2009.

[27] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The simos approach. IEEE Parallel Distrib. Technol., 3(4):34-43,1995.

[28] C. Kersey, A. Rodrigues, and S. Yalamanchili. A universal parallel front-end for execution driven microarchitecture simulation,. Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation Methods and Tools, p(p):25{32, 2012.

[29] R. Bedicheck. Simnow: Fast platform simulation purely in software,. In Hot Chips 16, Aug 2004.

[30] Intel. Qemu, a fast and portable dynamic translator. In USENIX 2005 Annual Technical Conf., pages 41-46, Apr 2005.

[31] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. in 22nd Annual International Symposium on Computer Architecture., pages 24-33, 1995.

[32] J.Wang, J. Beu, S. Yalamanchili, and T. Conte. 2012. Designing Configurable, Modifiable and Reusable Components for Simulation of Multicore Systems. In 3rd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS12). 472–476.

[33] William James Dally; Brian Towles (2004). "13.2.1". Principles and Practices of Interconnection Networks. Morgan Kaufmann Publishers, Inc. ISBN 978-0-12-200751-4.

[34] CAS Latency, accessed September 22, 2015, retrieved from "https://en.wikipedia.org/wiki/CAS_latency"

[35] Intel Corporation, Xeon Processor E7/E5 Family, accessed September 22, 2015, retrieved from "https://www-ssl.intel.com/content/www/us/en/processors"

[36] Transistor Count, accessed September 23, 2015, retrieved from https://en.wikipedia.org/wiki/Transistor_count

[37] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. Thread-level parallelism of desktop applications. Workshop on Multi-threaded Execution, Architecture and Compilation, 2000.

[38] J.K. Peacock, J.W. Wong and E.G Manning, Distributed Simulation using a Network of Processors, Computer Networks, Volume 3, Issue 1, February 1979, Pages 44 – 56

[39] Redhat Corporation, Red Hat Enterprise Linux 6, 6.3 Release Notes, accessed September 23, 2015, retrieved from "https://access.redhat.com/documentation/ en-US/Red_Hat_Enterprise_Linux/6/html/6.3_Release_Notes/"

[40] Open MPI: version 1.5.4, accessed September 23, 2015, retrieved from "https://www.open-mpi.org/software/ompi/v1.5/"

[41] E. Argollo, A. Falcon, P. Faraboschi, M. Monchiero and D. Ortega, COTSon: infrastructure for full system simulation, ACM SIGOPS Operating Systems Review, Volume 43 Issue 1, January 2009 Pages 52-61

[42] G. Loh, S. Subramaniam, and Y. Xie, "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration,", International Symposium on Performance Analysis of Software and Systems, pp. 53–64, 2009.

[43] J. Billington, S. Saboo, "An Investigation of Credit-based Flow Control Protocols,", Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and System & workshops, Article No. 34

[44] D.S. Henty, "Performance of Hybrid message-passing and Shared-memory Parallelism for Discrete Element Modeling," Proceedings of the 2000 ACM/IEEE conference on Supercomputing, Article No. 10.

[45] The NS-3 network simulator, accessed September 23, 2015, retrieved from "https://www.nsnam.org/"

[46] J. Chen, M. Annavaram, and M. Dubois. September 2009. SlackSim: A Platform for Parallel Simulations of CMPs on CMPs. ACM SIGMETRICS Performance Evaluation Review, volume 37 issue 2, pages 77–78.

[47] A.F. Rodrigues, K.S. Hemmert, K. Bergman, D. Bunde, E. Cooper-Balis, and K. Ferreira. 2012. Improvements to the Structural Simulation Toolkit. Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques, page 190-195.